SHELL SCRIPTS

UNIX シェルスクリプト

マスターピース 132

大角祐介

SB Creative

SHELL SCRIPTS

UNIX シェルスクリプト

マスターピース 132

大角祐介

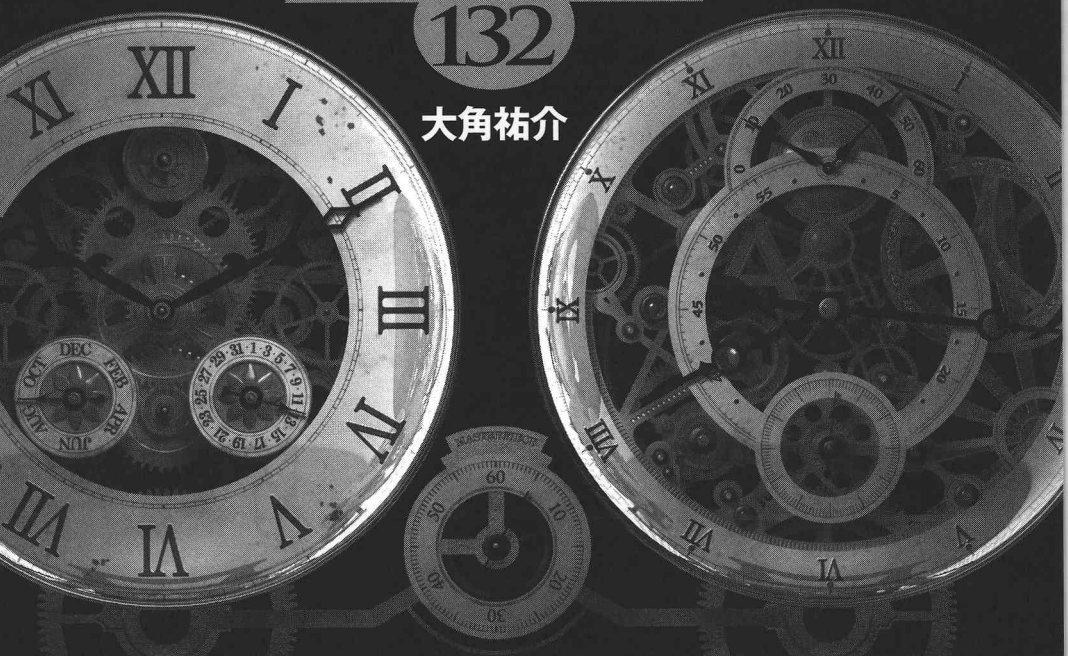
SB Creative

SHELL SCRIPTS

UNIX シェルスクリプト

マスターピース 132

大角祐介



SB Creative

はじめに

本書は、LinuxやFreeBSDなどのUNIXマシンでシステム開発・運用を行なっている方、あるいはこれからUNIXを学ぼうとしている方を主な対象とした、シェルスクリプトの実用サンプル集です。

シェルスクリプトはUNIXユーザにとって非常に身近なプログラミング言語であり、ログのデータ抽出やファイル操作、テキストファイルの加工・整形、プロセスの制御、サーバ構築のためのパッケージ管理など、利用されるシーンも数多くあります。UNIXで何か作業をしようとするならば、シェルスクリプトの知識は必須と言ってよいでしょう。

しかしいざシェルスクリプトを学ぼうとしても、単にコマンドを並べてスクリプトを書くだけのところからなかなか先に進めずにいるという方も多いことでしょう。

本書は、実用的なシェルスクリプトを書けるようになることを目的としています。そのためシェルスクリプトのテクニック的な解説に終始するよりも、UNIXの運用管理においてよく見られる問題を想定して、それを解決する実用サンプル例を紹介しています。いま読者が直面している、あるいはこれから直面するであろう問題に、きっと役に立つことと思います。

本書の内容が、読者の日々の作業に少しでも力になれば幸いです。

なお本書を執筆するにあたり、三宅英明(@mollifier)氏・伊能隆之氏のおふたりにレビューを行なっていただき、多くのご指摘・意見をいただきました。この場を借りまして、感謝の意を表します。

2014年5月
大角祐介

本書の読み方

シェルの確認方法

本書では、サンプル例はshもしくはbashで記述しています。shはどの環境でも存在しますが、bashはインストールされていない場合があるかもしれません。その場合には、APPENDIXの「09 bashのインストールについて」を参考にして、まずbashのインストールを行ってください。

普段読者が端末ウィンドウでの操作に利用しているシェルは、「ログインシェル」と呼ばれます。これは次のようにシェル変数SHELLをechoコマンドで表示することで確認できます。

ログインシェルの確認

```
$ echo $SHELL
/bin/bash
```

基本的にログインシェルは、FreeBSDの場合はtcshが、LinuxおよびMacの場合はbashが利用されることが多いようです。また読者の中には、高機能なシェルであるzshを使っている方もいるかもしれません。

ログインシェルと、シェルスクリプトを実行するシェルはそれぞれ別物ですから、これが違っていても気にする必要はありません。ただし、tcshなどのCシェル系をログインシェルとして利用している場合には、コマンドラインで普段行う操作とシェルスクリプトの文法が一致しないことがあるため、若干のとまどいがあるかもしれません。

ログインプロンプト

先ほどの図においてechoコマンドの前の$記号は、シェルのプロンプト記号です。これは、シェルが「いま、コマンド入力待ち状態ですよ。何かコマンドを入力してください」と入力を促す(prompt)ことからプロンプトと呼ばれています。なお、本書ではプロンプト直後でユーザが入力する文字を太字にしています。

シェルのプロンプトはユーザが自由にカスタマイズできるため、読者の環境により使われている記号は異なるかもしれません。しかし通常は、shやbashなどのBシェル系では一般ユーザは$、rootユーザは#が用いられます。そのため本書の実行例などでも、プロンプトが$となっていれば一般ユーザで、#となっていればrootユーザで実行するよう意図して記述しています。

なお、tcshなどCシェル系では、一般ユーザのプロンプトは%がよく使われます。その

ためFreeBSDでtcshを利用している場合などには、本書で例示しているプロンプトの$を%に読み替えてください。

シェルスクリプトの書き方

シェルスクリプトは単なるテキストファイルですから、読者が使いやすい、お好きなエディタを使って書いてください。一般的にはvi (vim) もしくはemacsが使われることが多いのですが、geditなどのGUIエディタを利用してもかまいません。

シェルスクリプトのファイル名は、何でもかまいません。ただし慣例的に拡張子を.shにすることが多いため、特別な理由がない限りそうしたほうがよいでしょう。また、ファイル名に日本語やスペースを入れるのは無用な混乱を招きますから、避けたほうが無難です。

なおファイルの拡張子がない場合、それがシェルスクリプトなのか通常の実行ファイルなのかわからないことがあります。このようなときは、fileコマンドを利用すると対象ファイルの形式を調べて表示してくれます。次の例では、scriptファイルは「shell script」と表示されていますから、これがシェルスクリプトであることがわかります。

fileコマンドでのファイル形式の確認

```
$ file script
script: Bourne-Again shell script text executable
$ file command
command: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked (uses shared libs), for GNU/Linux 2.6.18, stripped
```

シェルスクリプトを書いたら、実行する前に文法チェックを行う-nオプションを利用することをお勧めします。-nオプションを利用すると、シェルはスクリプトを1行ずつ読み込んで解釈を行いますが、実行はしません。そのため、書いたスクリプトに文法エラーがないかを実行前に確認することができます。

-nオプションを利用して文法チェック

```
$ sh -n script.sh
$                                          文法エラーがなければ、何も表示せずに終了する

$ sh -n script.sh
script.sh: line 6: syntax error: unexpected end of file ——— 文法エラーがあれば、エラーが表示される
```

シェルスクリプトの各種実行方法

シェルスクリプトを実行する際には、まずファイルに実行権限を付ける必要があります。これには次のように、chmodコマンドの実行ビットを立てる+xオプションを利用します。

ファイルに実行権限を付加する

```
$ chmod +x script.sh
```

chmodコマンドでは、パーミッションを3桁の8進数で設定することもできます。詳しくは「man chmod」として、chmodコマンドのマニュアルを読んでみてください。

コマンドラインからシェルスクリプトを実行する際には、UNIXでは一般的にカレントディレクトリにパスが通っていないことが普通であるため、明示的にパスを指定してスクリプトファイルを実行する必要があります。この際にはフルパス指定で実行してもよいのですが、カレントディレクトリを意味する./を頭に付けるパターンが通常はよく使われます。

スクリプトを実行する（/home/user1/binがカレントディレクトリと仮定）

```
$ /home/user1/bin/script.sh ―――― フルパス指定で実行

$ ./script.sh ―――― カレントディレクトリを意味する./を頭に付けて実行
```

また、集計処理や監視などで定期的に実行するシェルスクリプトは、コマンドラインから直接実行せず、cronで自動起動させるシーンも多く見られます。cronを利用する例については、APPENDIXの「04 cronによるスクリプト実行について」を参照してください。

CONTENTS

CHAPTER 03 ファイル処理

CHAPTER 04　日付処理

CHAPTER 05　ネットワーク

CHAPTER 06　テキスト処理

CHAPTER 07 シェルの機能を使いこなす

CHAPTER 08 制御構文のサンプル

CHAPTER 09 サーバ管理

CHAPTER 10 bash

APPENDIX 追加情報

CHAPTER

01

ユーザインタフェース

この章では、シェルスクリプトでのコマンドオプションの指定方法や、キーボード入力を扱う端末処理、メッセージのカラー表示、テキストベースでのダイアログボックスの利用方法などについて紹介します。
ユーザにログインIDやパスワードを入力してもらいたいときや、メニューやカレンダーを表示して選択させる対話型のシェルスクリプトを作りたい時には、これらのサンプル例が役に立つことでしょう。

No. 001 コマンドオプションの処理をする

利用コマンド
getopts, case, shift

キーワード
オプション, フラグ, コマンドライン引数

いつ使うか
スクリプト内でオプション(-aなど)を解析して動作を変えたいとき

実行例

```
$ ./getopts.sh -a -p '=====sep=====' /home/user1/docs
.  ..  a.txt  readme.txt
=====sep=====
```

スクリプト

```
#!/bin/sh

# -aオプションが付加されたかどうかのフラグ変数a_flagと、
# -pオプションのセパレータ文字列を定義する
a_flag=0                                              ❶
separator=""

while getopts "ap:" option                            ❷
do
  case $option in
    a)
      a_flag=1                                        ❸
      ;;
    p)
      separator="$OPTARG"                             ❹
      ;;
    ¥?)                                               ❺
      echo "Usage: getopts.sh [-a] [-p separator] target_dir" 1>&2
      exit 1
      ;;
  esac
done

# オプション指定を位置パラメータから削除する
shift $(expr $OPTIND - 1)                             ❻
path="$1"                                             ❼

# -aオプションが指定されたかどうかを、シェル変数a_flagの値で判断する
```

```
if [ $a_flag -eq 1 ]; then
  ls -a -- "$path"
else
  ls -- "$path"
fi

if [ -n "$separator" ]; then
    echo "$separator"
fi
```

❽

❾

解説

UNIXのコマンドでは、実行時にさまざまなパラメータを指定することがよくあります。例えばファイルをコピーするcpコマンドは、デフォルトでは既存ファイルがあった場合でも確認なしに上書きコピーをしてしまいますが、「-i」を付けると、次のように上書き前に確認をするようになります。

上書き前に確認するようになった

```
$ cp -i old.txt new.txt
cp: overwrite `new.txt'?
```

上記の-iというのが**オプション**です。また、この例では「-i old.txt new.txt」というコマンドに与えているパラメータをまとめて、**コマンドライン引数**と呼びます。

このように、オプションを利用すると、同一のコマンドでもデフォルト状態から動作を変えることができたり、特定のパラメータを指定したりすることができるようになります。

同様に、シェルスクリプトを作る際にも、オプションを指定してデフォルト動作を変えたい場合があります。オプションは通常-(ハイフン)で始まるため、コマンドライン引数を自分で解析して処理分岐を書けば実現できますが、これはなかなか面倒な作業です。それよりも、オプション解析をしてくれる専用のビルトインコマンド、**getoptsコマンド**を用いるのが便利です。

このスクリプト例は、現在のディレクトリの中にあるファイル名一覧を表示するだけのものですが、オプションによって多少動きが変わるようにできています。-aオプションを付けることで隠しファイル(ドットで始まるファイル)も表示できるようにしており、また-pオプションを付けることで最終行にセパレータとして表示する文字列を指定できるようにしています。

さらに、デフォルトではカレントディレクトリの中身を表示しますが、コマンドライン引数としてディレクトリを指定すれば、それを対象のディレクトリとするようにしています。

では肝心のgetoptsコマンドの使い方ですが、書き方がちょっとわかりづらいので、以下少し丁寧に見ていきましょう。

まず❶では、-aオプションが指定されたかどうかを判断するシェル変数a_flagを宣言しています。このような変数はフラグ(旗)と呼び、0で初期化しておき、設定されたら1としておくのが一般的です。

❷で実際にgetoptsでオプションを解析しています。オプションに使う文字はgetoptsの引数として並べ、さらにそのオプション自体が引数を取るものは、コロンを付けて指定します。つまり、ここでは"ap:"としていますが、これは「aオプションとpオプションを利用する。そしてpオプションは引数を受け取る」という意味です。getoptsはオプションを先頭から順番に解析していくため、❷のようにwhileの対象とすれば、指定されたすべてのオプションを順番に処理できます。getoptsの2つ目の引数にはシェル変数(ここではoption)を指定します。ユーザが入力したオプションはこのシェル変数optionに代入されるため、以下のcase文で変数optionの値により分岐することで、入力されたオプションごとの処理が行えます。

なお、このようにgetoptsを利用する際は、「**while文の条件式としてgetoptsを書いて、そのwhileループ内部のcase文で判断する**」というのがセオリーですので、この書き方は丸ごと覚えてしまうとよいでしょう。

この例ではまずaオプションが指定された場合はa_flagを1として、「aオプションが指定された」とフラグに設定しています(❸)。一方、引数を受け取るオプション(つまりgetoptsにてコロン付きで指定したオプション)では、その受け取った引数はOPTARGというシェル変数に入っています。❹ではこの値を、separatorというシェル変数に代入しています。これは最後の出力時に用いています。

ここまででマッチしなかったオプションを、❺で処理しています。getoptsでは無効なオプションが指定された場合は"?"が代入されるため、これをcase文で処理しているのです。この❺のcase処理にきた場合はオプションの指定にエラーがあるため、次のようにエラーメッセージを出力して終わりにします。

不正なオプションを指定した場合

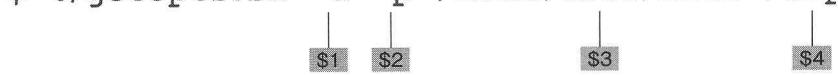
```
$ ./getopts.sh -a -z
./getopts.sh: illegal option -- z
Usage: getopts.sh [-a] [-p separator] target_dir
```

なお、上記の実行例をよく見ると、このスクリプトでは書かれていない"illegal option"というエラーを出力しています。これはgetoptsコマンド自体が持つ機能で、定義しなかったオプションが指定された場合は自動で出してくれるのです。

続いて、❻は理解がなかなか困難です。これは慣用句かイディオムのようなものだと思って丸覚えしてしまってもよいのですが、きちんと理解したい方のために以下に解説します。

この行は先に目的から言うと、コマンドライン引数を**位置パラメータ**($1, $2, ...)として正しく扱いたいために実行しています。本来、位置パラメータは、オプションもパラメータもひっくるめて $1, $2, $3 ,として格納されています。

コマンドライン引数の位置パラメータ



getoptsコマンドでオプション解析が終わった後には、シェル変数**OPTIND**は「次に処理する位置パラメータの番号」を指しています。つまり上記の例では、"4"になっています。

ここで、OPTINDから1を引いた値で**shiftコマンド**を実行することで、オプション部分を無視した「本当のコマンドライン引数」を順番に$1, $2, $3 として扱うことができます。つまり、shift $(expr $OPTIND - 1)とすることで、$1として/tmpが取り出せるようになるのです。

❼でこの仕組みを利用しています。このスクリプトでは、オプションではなく通常のコマンドライン引数として指定したディレクトリも対象とできる作りになっています。この際、オプションがいくつ指定されるか、あるいは1つも指定されないかが事前にはわからないので、オプション部分をshiftで「追い出し」てから改めて$1を取得することで、通常のコマンドライン引数を取得できます。

❽ではオプション-aが指定されたかどうかを、❸で設定したa_flagの値で処理を分岐し、lsコマンドでディレクトリの内容を表示しています。なおここで、"$path"の前に--というハイフン2つのオプションを指定しています。これは、もし$pathがハイフンで始まるファイル名だった場合に、オプションと扱われないように付加しているものです。詳しくは、APPENDIXの「UNIXコマンドのオプションについて」を参照してください。

最後に❾で、-pで指定されたセパレータを表示します。セパレータが指定されなかった場合はシェル変数separatorは空文字列になっています。それを**testコマンド**の**-n演算子**で判断し、セパレータ文字列が入っている場合のみ出力されることになります。testコマンドの詳細については、P.109を参照してください。

注意事項

・引数を持つオプションを複数使いたい場合は、次のようにコロン付きで並べます。

```
getopts "ap:x:z:"
```

上記の例では、aオプションは引数を取らず、pオプション、xオプション、zオプションは引数を取ります。

No. 002 キーボードから Ctrl + C が入力されたときに、現在の状態を出力してから終了する

利用コマンド

trap, exit, curl, sleep

キーワード

シグナル, トラップ, 終了

いつ使うか

長い時間のかかる処理や無限ループの処理内で、ユーザが途中終了させるためにキーボードで Ctrl + C を入力したとき、すぐ終了させずに何らかの処理を行いたいとき

実行例

```
$ ./sigint.sh http://www.example.org/
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  1270  100  1270    0     0   2903      0 --:--:-- --:--:-- --:--:--  9921
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  1270  100  1270    0     0   5384      0 --:--:-- --:--:-- --:--:-- 10948
^C ───────────────── ここで Ctrl + C を押した
Try count: 2
```

スクリプト

```
#!/bin/sh

count=0
trap ' echo
       echo "Try count: $count"          ❶
       exit ' INT

while :
do
  curl -o /dev/null $1
  count=$(expr $count + 1)               ❷
  sleep 1
done
```

解説

このスクリプトは、コマンドライン引数で指定されたURLに、1秒に1回curlコマンドでアクセスし続けるものです。キーボードから Ctrl + C が入力されると動作を止め、それまでにアクセスした回数を「Try count: 2」のように表示します。

現在の端末で、フォアグラウンドで動作しているシェルスクリプトは、Ctrl + C で強制終了させることができますが、このときの動きは、以下のようになっています。

1) Ctrl + C を入力すると、シェルスクリプトのプロセスに**SIGINT**というシグナルが送られる。
2) SIGINTを受け取ったシェルスクリプトのプロセスは、キーボードからの割り込み(Interrupt)を通知される。
3) キーボードからの割り込み(Interrupt)のデフォルト動作は、プロセスを終了することであるため、シェルスクリプトは実行終了する。

ここで出てきた**シグナル**とは、実行中のプロセスに対してさまざまな動作を指示することができる仕組みのことです。シグナルには多くの種類があり、それぞれ機能も違います。キーボードから Ctrl + C を入力された際には、シグナル番号2のSIGINTが送られます。

シグナルの一覧を確認したい場合は、次のように**killコマンド**に**-lオプション**を付けるとシステムのシグナル一覧を表示することができます。

シグナル一覧はkillコマンドの-lオプションで確認

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
 ...
```

さてSIGINTを受け取ったプロセスは、通常はそこで終了します。しかしシェルスクリプトでは、シグナルを受け取った際の動作を**trapコマンド**で制御することができます。trapコマンドの書式は次のように、"内に行いたい処理、後ろに制御したいシグナル名を記述します。この例では、Ctrl + C が押されると現在時刻を表示してから**exitコマンド**でシェルスクリプトを終了します。

```
trap ' date; exit ' INT
```

行いたい処理　シグナル

trapコマンドがよく使われる例としては、SIGINTなどの終了すべきシグナルを受信した際、その場ですぐに終了せずに、ログを出力したり現在の状態を表示してから終了するというようにデフォルトの動作を上書きするケースです。

サンプル例は、1秒に1回**curlコマンド**でWebサイトからダウンロードを行い(❷)、通信が正常かを目で見て確認するスクリプトです。このプログラムは無限ループとなっているため Ctrl + C を入力するまで終了しません。そしてSIGINTを❶の部分でtrapしているため、 Ctrl + C を入力した際には、全部で何回curlを実行したか、現在の状態を変数countで出力してから終了するようにしているものです。

注意事項

・FreeBSDではcurlは標準でインストールされていません。そのため次のように、**fetchコマンド**で代用するとよいでしょう。

```
fetch -o /dev/null $1
```

・次のように実行するコマンドを空にすれば、 Ctrl + C を無視して、キーボードからは終了できないプロセスを作ることもできます。この場合は、別の端末からkillコマンドで**TERM**シグナル(killコマンドがデフォルトで送信するシグナル)を送れば終了することができます。

```
trap '' INT
```

関連項目

087 スクリプト実行中にシグナルを受け取って、現在の実行状態を出力する
088 HUPシグナルを受け取って実行中に設定ファイルを読み込みなおす
089 異常終了してもゴミが残らないよう、終了前に作業ファイルを消去して後始末を行う

No. 003 キーボードからユーザのキー入力を取得して、変数の値として利用する

利用コマンド
read

キーワード
標準入力, キーボード

いつ使うか
キーボードから入力された値を用いて、対話的に処理を行いたいとき

実行例

```
$ ./read.sh
Enter your ID: guest
Now your ID is guest
```

入力された値を表示する

スクリプト

```
#!/bin/sh

echo -n "Enter your ID: "
read id ―――――――――――――――――❶

echo "Now your ID is $id" ――❷
```

解説

このスクリプトは、ユーザにキーボードから自分のIDを入力してもらい、その値をシェルスクリプト内で変数として利用してIDを表示するものです。

シェルスクリプトでキーボードから入力された値を取得するには、**readコマンド**を用います。readコマンドは❶のように、値を入れたいシェル変数を引数にとります。つまりこの例では、シェル変数idにユーザがキーボードから入力した値が代入されます。こうしてシェルスクリプト中で、ユーザからの入力を取得することができます。

続いて❷で、キーボードから入力された値を用いてメッセージを表示しています。変数idには先ほど入力されたIDが入っています。この後に、入力されたIDに応じた処理を書いていくことができるでしょう。

なお、readコマンドで複数の入力値をとりたい場合は、次のようにreadコマンドの後ろに複数の変数をスペースで区切って並べます。

```
echo -n "Enter your ID NAME NUMBER: "
read id name number
```

こうすれば、ユーザが次のように入力すると、変数idにはguest、変数nameにはSato, 変数numberには341が入ることになります。

```
Enter your ID NAME NUMBER: guest Sato 341
```

もっとも、複数の値を1行で入力するのは、ユーザにとってなかなか面倒です。実際にはあまりまとめて入力させず、次のように値を1つ1つ入力するほうが利便性が高い場合が多いでしょう。

リスト1 まとめて入力するより個別に入力するほうが使いやすい

```
echo -n "Enter your ID: "
read id
echo -n "Enter your NAME: "
read name
echo -s "Enter your NUMBER: "
read number

echo "Now your ID is $id."
echo "NAME: $name, NUMBER: $number
```

注意事項

・readコマンドを実行すると、ユーザからの入力を待つためにスクリプトは一時停止します。そのため、次のように単にスクリプトを一時停止したい際にもreadコマンドを利用できます。この例では入力値は不要なので、代入するシェル変数はダミーとしています。

```
#!/bin/sh

echo "Input any key to continue..."
read dummy
echo "Script start."
```

・Macで改行させないメッセージを出力するecho -nを利用する際は、P.15を参照してください。

関連項目

004 パスワード入力の際に、ユーザのキー入力を表示しないようにする
005 ユーザのキー入力を1文字だけ取得する（入力時にEnterを不要にする）
007 選択式メニューを表示して、入力された数値の処理を実行する

No. 004 パスワード入力の際に、ユーザのキー入力を表示しないようにする

利用コマンド
stty, read, wget, curl

キーワード
パスワード

いつ使うか
ユーザからパスワードを入力してもらう際、入力された文字列を画面に表示しないようにしたいとき

実行例

```
$ ./pass_wget.sh
Password:                    ここで入力しても何も表示されない
```

スクリプト

```
#!/bin/sh

username=guest
hostname=localhost

echo -n "Password: "
# エコーバックをOFFにする (-echo)
stty -echo                                    ❶
read password                                 ❷
# エコーバックをONにする (echo)
stty echo                                     ❸

echo

# 入力されたパスワードでダウンロードを行う
wget -q --password="$password" "ftp://${username}@${hostname}/filename.txt"
                                              ❹
```

解説

このスクリプトは、ユーザにキーボードからパスワードを入力してもらい、ftpサイトからファイルをダウンロードするものです。パスワード入力時には画面に入力値が表示されない工夫をしています。

ユーザのキー入力を**readコマンド**で取得する際、入力された文字はそのまま画面に表示されます。当たり前のことのように思われるかもしれませんが、これはエコーバックと

呼ばれる機能で「入力された文字を画面に表示する」ように端末が設定されているからです。

一方、ユーザにパスワードを入力してもらうプログラムでは、画面を盗み見られる危険性があります。そのための対策として、入力された文字が画面に表示されないよう**エコーバックをオフ**にするのが普通です。

エコーバックをしないように設定するには、❶のように端末の設定を変更する**sttyコマンド**で-echoを指定します。これで端末のエコーバックはなくなり、入力した文字が画面に表示されないようになります。入力されたパスワードは、readコマンドを用いてシェル変数passwordに代入しています(❷)。

このままではエコーバックしない設定が残ってしまうため、パスワード入力が終わったところで、❸のようにsttyコマンドでechoを指定して端末を元の状態に戻します。こうすれば端末は再びエコーバックするようになり、入力した文字がそのまま表示されます。

なお、端末について詳しくは、APPENDIXのP.380を参照してください。

サンプル例では、入力されたパスワードを使って、**wgetコマンド**でftpサーバからファイルをダウンロードしています。wgetコマンドでユーザ名とパスワードを入力する際は、❹のように**--passwordオプション**の引数にパスワードを指定し、ダウンロード先のホスト名の頭に@を付けてユーザ名を入力します。

またwgetコマンドには余計な出力をしない**-qオプション**(quietモード)を利用し、ファイルのダウンロードのみ行うように指定しています。

注意事項

- エコーバックの機能がピンとこない方は、端末で次のコマンドを直接実行してみましょう。

```
stty -echo
```

こうすると、以後はコマンドを入力しても画面には表示されませんが、Enterを叩けばコマンドの実行結果は表示されます。元に戻すには、(表示されませんが) stty echoとタイプしてEnterを押せば、再びキー入力は表示されます。

- Macでは、標準でwgetコマンドがインストールされていません。代わりに**curlコマンド**を用いて次のように書くことができます。

```
curl -s -u "${username}:${password}" -O "ftp://${hostname}/filename.txt"
```

関連項目

003 キーボードからユーザのキー入力を取得して、変数の値として利用する

No. 005 ユーザのキー入力を1文字だけ取得する（入力時にEnterを不要にする）

利用コマンド

stty, case, dd

キーワード

キーボード, 入力, 改行, Enter

いつ使うか

キーボードから入力してもらう際に、1文字入力されたら処理を続行し、Enterの入力を不要としたいとき

実行例

```
$ ./getchar.sh
Type Your Answer [y/n]: y
Input: YES
```

スクリプト

```
#!/bin/sh

echo -n "Type Your Answer [y/n]: "

# 現在の端末設定をシェル変数tty_stateにバックアップしてから、
# 端末をraw設定する
tty_state=$(stty -g) ―――――――――――――――――――――――――――❶
stty raw ―――――――――――――――――――――――――――――――――❷
# キーボードから1文字読み込む
char=$(dd bs=1 count=1 2> /dev/null) ―――――――――――――――――❸
# 端末設定を元に戻す
stty "$tty_state" ―――――――――――――――――――――――――――――❹

echo

# 入力された文字により処理を分岐する
case "$char" in
  [yY])
    echo "Input: YES"
    ;;
  [nN])
    echo "Input: NO"
    ;;
  *)
    echo "Input: What?"
    ;;
esac
```

❺

解説

このスクリプトは、ユーザにYes/Noの問い合わせを行い、キーボードから入力された文字でYesかNoかを判断してメッセージを表示するものです。

スクリプトの実行中、「続行しますか?」と問い合わせるなど、何かしらの入力をユーザにしてもらいたい場合があります。そのような場合はreadコマンドを使うことでユーザのキー入力を取得することできますが、**readコマンドは必ず改行が必要**です。つまり、ユーザはキーボードから文字を入力した後にEnterを押さなければいけません。

しかし単純なYes/Noの問い合わせなどでは、Enterを押さずとも、yもしくはnを押しただけで処理を進めたい場合があります。このような場合は端末(ターミナル)の状態を設定する**sttyコマンド**で端末を**rawモード**にすると、キーバッファ処理が行われないため、1文字ごとに処理を行うことができます。

この例では、改行しないメッセージを**echo -n**で表示した後に、現在の端末の設定をstty -gコマンドの出力結果を元に保存します(❶)。stty -gの出力は次のようになっており、現在の端末の設定が保存されています。

「現在の端末の設定」を表示してみる

```
$ stty -g
2d00:5:bf:ca3b:3:1c:7f:15:4:0:1:0:11:13:1a:ff:12:f:17:16:
ff:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0
```

この後rawモードにするときに現在の端末設定がクリアされてしまうため、後で設定を元に戻せるよう、現在の設定を一時的にシェル変数tty_stateに保存しているのです。

続いて❷で、stty rawコマンドによって端末をrawモードに切り替えます。こうすると実際のキー入力の値は、次の**ddコマンド**に直接渡されます。❸のddコマンド内では、入出力のブロックサイズを1(bs=1)、入力から出力へとコピーするブロック数を1(count=1)とすることにより、入力された文字をシェル変数charにセットしています。

また、ddコマンドの実行メッセージはこの場合には不要なため、/dev/nullへリダイレクトして消しています。

このままでは端末がrawモードのままとなってしまうため、スクリプト実行前の状態に戻す必要があります。❹のように、sttyコマンドを使ってstty "$tty_state"とセットすれば、ユーザの端末状態はrawモードを抜けて最初に保存しておいた端末の状態へ戻ります。

入力された値はシェル変数charに入っているため、❺でcase分岐させています。入力された文字がyもしくはYならばYesの処理、nもしくはNならばNoの処理を行えばよいでしょう。

Macのechoコマンド

Macのechoコマンドで改行させないようにするのは少々厄介なため、以下に少し詳し

く解説します。

Macのechoコマンドは、コマンドラインで直接使う場合、改行させない-nオプションが機能します。しかし、シェルスクリプトの中でecho -nと書いて、スクリプトとして実行すると、オプションのつもりで指定した"-n"が文字列としてそのまま出力されてしまいます。

これは、echoコマンドがシェルのビルトインコマンドとして実行されていることが原因です。シェルスクリプトで利用するコマンドには、「外部コマンド」と「シェルのビルトインコマンド」の2種類があります。外部コマンドとは、/bin/echoのように実行ファイルが存在しているコマンドです。一方、シェルのビルトインコマンドとは、シェル自体が内部で持っているコマンドで、実行ファイルが存在しません。

このサンプルのようにechoと書くと、シェルのビルトインのほうのechoコマンドが利用されます。Macでは一般的にログインシェルにbashが指定されているため、コマンドラインではbashのシェルビルトインのechoコマンドが実行されます。これは-nに対応しているため、正しく「改行しない」動作をします。

一方、スクリプト中でechoと書くと、bashではなくshのシェルビルトインのechoコマンドが実行されます。Macのshシェルビルトインのechoコマンドは、改行させないオプション-nに対応していないため、指定しても引数として解釈され、"-n"がそのまま出力されます。

そのためMacで改行したくないメッセージをechoコマンドで表示させるときは、次のように**外部コマンドのecho**を使うように書きます。

```
/bin/echo -n "Type Your Answer [y/n]: "
```

あるいは次のようにprintfコマンドを使うという方法もあります。printfコマンドは¥nで明示的に指定しないと改行が入らないので、改行なしで出力できます。

```
printf "Type Your Answer [y/n]: "
```

関連項目

003　キーボードからユーザのキー入力を取得して、変数の値として利用する

No. 006 ファイルから読み込んで処理をしているときに、キーボードからの入力を行う

利用コマンド
tty, read

キーワード
標準入力, キーボード, 外部ファイル

いつ使うか
readコマンドでファイルから読み込んでいる処理の中で、さらにreadコマンドでキーボードからの入力を得たいとき

実行例

```
$ ./read-redirect.sh
Input Target Directory:
/home/user1/mydir ―――――――― キーボードからディレクトリを入力

data1.txt  data2.txt
```

スクリプト

```
#!/bin/sh

tty=`tty` ―――――――――――― ❸
while read question ―――――― ❶
do
  echo $question
  read dir < $tty ―――――――― ❹
  echo "Command: ls $dir"
  ls $dir
done < question.txt ―――――― ❷
```

解説

このスクリプトは、question.txtに書かれた質問を1行ずつ順番に表示しながら、キーボードから入力されたディレクトリ内のファイルをlsコマンドで表示するものです。

readコマンドは標準入力(キーボードからの入力)を読み込んでその値をシェル変数に代入しますが、**ファイルの中身を1行ずつ読んでシェル変数に代入**することもできます。その際はまず❶のようにwhileの条件式にreadコマンドを指定すると同時に、❷のようにdone直後にリダイレクトを書くことでwhileループ全体に入力リダイレクトします。こうすれば、ファイルの中身をreadコマンドが1行ずつ読み込んでシェル変数questionに代入してくれるので、1行ずつ処理ができます。

この例では、question.txtにはリスト1のような内容が書かれています。ディレクトリ

名を尋ねる質問文を外部ファイルにしておき、ユーザからの入力値でそのディレクトリをlsして、中にあるファイルを表示しようとしています。

リスト1 スクリプトで使用する質問ファイル

```
Input Target Directory:
```

しかし、こうしてreadコマンドにリダイレクトをしている処理の中で、さらにreadコマンドを使ってキーボードからの入力を得たい場合、そのままreadコマンドを実行しても、すでに標準入力はファイルとなってしまっているため正しく動きません。

そこでこの例では、❸のように事前に端末情報を**コマンド置換**により保存しています。シェルスクリプトでは、コマンド行を`(バッククォート)で囲むと、その部分はコマンド実行された結果に置き換わり、これをコマンド置換と呼びます。この例では**ttyコマンド**の実行結果がシェル変数ttyに代入されます。コマンド置換については、P.51も参照してください。

そして標準入力がリダイレクトされている処理内でも、❹のように端末情報を直接リダイレクトすることで、readコマンドにリダイレクトしているwhileループ内でも、正しくキーボードからの入力を得ることができます。

注意事項

・サンプル例に出てきたttyコマンドの出力は、具体的には次のように「/dev/pts/0」など、現在利用している仮想端末のデバイスファイル名となっています。

⬇ttyコマンドの出力

```
$ tty
/dev/pts/0
```

❹では、この仮想端末をリダイレクトすることで、キーボードからの入力を明示的に指定できたわけです。仮想端末とデバイスファイルについては、APPENDIXの「端末(ターミナル)とは」(→P.390)も合わせて参照してください。

関連項目

003 キーボードからユーザのキー入力を取得して、変数の値として利用する

No. 007 選択式メニューを表示して、入力された数値の処理を実行する

利用コマンド
read, case

キーワード
標準入力, キーボード

いつ使うか
キーボードから入力された値を用いて、対話的に処理を行いたいとき

実行例

```
$ ./select.sh
Menu:
1) list file
2) current directory
3) exit
2
/home/user/shell_script

Menu:
1) list file
2) current directory
3) exit
3
```

スクリプト

```
#!/bin/sh

while :
do
  echo "Menu:"
  echo "1) list file"
  echo "2) current directory"
  echo "3) exit"

  read number ―――――――――――❶
  case $number in ―――――――――――❷
    1)
      ls
      ;;
    2)
      pwd
      ;;
    3)
```

```
      exit
      ;;
    *) ―――――❸
      echo "Error: Unknown Command"
      ;;
  esac

  echo
done
```

解説

これは番号付きメニューを表示してユーザに値を入力してもらい、指定された番号の処理を実行するものです。

このようなメニュー付きのスクリプトを作る場合は、番号と処理内容をechoコマンドで表示してからユーザの入力を**readコマンド**で取得して、その入力内容を**case文**で判断して分岐させる手法が簡単でよく使われます。readコマンドは❶のように、シェル変数を引数として指定することで、標準入力(ここではキーボードからの入力)をシェル変数に代入できます。この例では、シェル変数numberにユーザがキーボードから入力した値が入ることになります。

続いて、❷のcase文で入力されたメニュー番号ごとの処理を行っています。この例では、1が入力されるとカレントディレクトリのファイルをlsコマンドで表示、2が入力されるとカレントディレクトリをpwdコマンドで表示、3が入力されるとexitコマンドで終了、としています。入力値による分岐はif文でも記述できますが、この例のようにシェル変数の値で分岐させる場合は、case文を使ったほうが利便性と保守性が上がります。コマンド実行後には元のメニューに戻るようにするため、全体をwhile文の無限ループとしています。

なお、ユーザからの入力値には、数値が入力されることを想定しているのに文字列が入力される、など予想しないものが入力される場合があります。❸のようにcase文の最後に*を用いると、それまでの条件にマッチしなかった値の処理が行えます。スクリプトが意図しない動作を引き起こさないように、このようなエラー処理を行うことを忘れないようにしましょう。

関連項目

003 キーボードからユーザのキー入力を取得して、変数の値として利用する
004 パスワード入力の際に、ユーザのキー入力を表示しないようにする
005 ユーザのキー入力を1文字だけ取得する(入力時にEnterを不要にする)

No. 008 表示文字の色などを変える

利用コマンド
echo

キーワード
端末, カラー

いつ使うか
メッセージを表示する際に、文字の色を変えたり反転させるなどして注意を引きたいとき

実行例

```
$ ./color.sh
Script Start.
 Important Message ―――――――― 表示色が変わっている
Script End.
```

スクリプト

```
#!/bin/sh

echo "Script Start."

# 背景をグレー (47)、文字色を赤 (31) にする
echo -e "¥033[47;31m Important Message ¥033[0m" ―――――――― ❶

echo "Script End."
```

解説

このスクリプトは、"Important Message"という文字列を、赤色で表示するものです。

シェルスクリプトで文字列を表示するには**echoコマンド**を用います。この際、メッセージの一部に注意を引きたい場合など、文字列の一部を目立たせたいことがあります。このような場合には、-eオプションを指定して**エスケープシーケンス**という制御コードにより、表示される文字に色を付けることができます。

エスケープシーケンスの書き方は、次のようにエスケープを表す**¥033[**と**m**で挟んだ中に色のパラメータを指定します。

書式 エスケープシーケンスの書き方

¥033[パラメータm 表示する文字列 ¥033[0m

例えば、単純に文字の色を赤(31番)にしたい場合は、次のようになります。

```
¥033[31m 表示する文字列 ¥033[0m
```

パラメータを複数指定する場合は、;(セミコロン)で区切って並べて書きます。例えば❶では、パラメータとして31番(文字色を赤)と47番(背景色を白)をあわせて指定しています。

文字色の指定をリセットするには、0番を指定して¥033[0mと書きます。これをしないと、それ以降の文字がすべて色指定されてしまうため、❶のように文字列の最後に置いておくのがよく使われる書き方です。

色指定の番号は次のとおりで、30番台がForeground Color(文字色)、40番台がBackground Color(背景色)です。

色を指定するときの値

	黒	赤	緑	黄	青	紫	水	白
文字	30	31	32	33	34	35	36	37
背景	40	41	42	43	44	45	46	47

これらの番号は、Linuxならば「man console_codes」でFreeBSDならば「man screen」で表示される、コンソールのエスケープシーケンスのマニュアルに記載されています。

注意点

- エスケープシーケンスを駆使すると、さまざまな色を用いて凝ったことをすることもできますが、ユーザが利用している端末(ターミナル)によっては、単に見にくいだけの結果となることも十分考えられます。色による強調は、あくまで補助的なものと考えたほうがよいでしょう。
- 環境によっては、echoコマンドを使う際に、エスケープシーケンスを解釈する-eオプションが必要ない場合があります。MacOS Xのshや、Ubuntuのように/bin/dashがshとなっているLinuxがこれに該当します。これらの環境では、次のように-eオプションなしでechoすると、色が付きます。

```
echo "¥033[47;31m Important Message ¥033[0m"
```

No. 009 カレンダーで選んで特定の日付のログファイルを削除する

利用コマンド
dialog, awk, rm

キーワード
ダイアログ, カレンダー , 対話型, 選択

いつ使うか
日付を指定する際に、対話的にカレンダーを表示して選択したいとき

実行例

```
$ ./dialog-calendar.sh
```

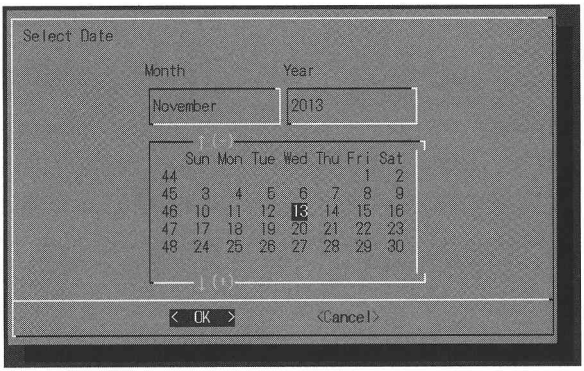
スクリプト

```
#!/bin/sh

LOG_DIR=/myapp/ap1/log

# dialogコマンドでカレンダー出力
# 選択日付は標準エラー出力にでるため、一時ファイルへリダイレクトする
dialog --calendar "Select Date" 2 60 2> cal.tmp ―――――❶

# カレンダー機能では 日/月/年 形式で出力されるため、これを
# 年月日に整形する
date_str=$(awk -F / '{print $3$2$1}' cal.tmp) ―――――❷

# キャンセルされた場合はテンポラリファイルを削除して終了
if [ -z "$date_str" ]; then
  rm -f cal.tmp
  exit
fi ―――――❸

rm -i ${LOG_DIR}/app_log.$date_str ―――――❹

# テンポラリファイルを削除する
rm -f cal.tmp
```

解説

このスクリプトは、カレンダーを表示して日付を選択してもらい、その日付をファイル名に持つログファイルを削除するものです。このようなケースではdialogコマンドを使うことにより、さまざまな対話型インタフェースを持つシェルスクリプトを作ることができます。実行すると、次のような表示になります。

dialog-calendar.sh実行中の様子



なおdialogコマンドのインストール方法は、APPENDIXのP.405を参照してください。
dialogコマンドのカレンダー機能を使用する場合は、次のような書式で起動します。

書式　dialogコマンドでカレンダーを利用する

```
dialog  --calendar text height width
```

text　：表示するテキストメッセージ
height：テキストメッセージの行数
width　：ダイアログボックスの横幅

❶では、「Select Date」の文字列を2行ぶんの高さで表示し、ダイアログボックスの幅を「60」で指定してあります。

カレンダーから日付を選択するには、Tabでフォーカスを移動してEnterで選択します。[Month][Year]のカラムではカーソルキーの上下で月と年を選択できます。真ん中のカレンダー部では、カーソルを矢印キーで動かすか、あるいはviエディタのようにh、j、k、lでもカーソルを動かして選択できます。なお、カレンダーの左端にある数字は、1年のはじめから数えて第何週かを示す週番号です。

ダイアログボックスをキャンセルしたいときは、下部の<OK><Cancel>カラムにTabで移動して、<Cancel>をEnterで選びます。

dialogコマンドでの選択結果は標準エラー出力へ表示されるため、この値を後でrmコマンドのパラメータとして使うためにテンポラリファイルへリダイレクトします。❶では標準エラー出力をテンポラリファイルcal.tmpへと出力するために、2> cal.tmpと書いています。

dialogコマンドの出力する日付を処理するとき、このスクリプトでは❷でちょっとしたテクニックを使っています。これはdialogコマンドのカレンダー機能の出力が、次のように日/月/年となっているためです。

```
13/11/2013
```

日本人からは奇妙に見える書き方ですが、この形式は西欧では比較的よく見られるポピュラーな書式です。これを日本式の年月日形式になおすには、/(スラッシュ)をセパレータとして3つの要素に分けて、それを逆順に出力すればよいわけです。❷はこのために、**awkコマンド**のオプションを-F /として/(スラッシュ)をセパレータとして指定しています。結果としてこのawkコマンド内では$1に13、$2に11、$3に2013という値が入ります。これを20131113という文字列とするため、$3$2$1と逆順に出力しています。

❸は、ダイアログでキャンセルされた場合に何もせず終了するための処理です。カレンダー選択時に<Cancel>が押されるとcal.tmpは空っぽとなるため、ここでのシェル変数date_strの値も空文字列となります。そのため**testコマンド**の**-z演算子**で空文字列かどうかを判断し、空ならば一時ファイルcal.tmpを**rmコマンド**で削除してからスクリプトを終了します。

❹で、日付指定されたファイルを削除しています。ここでは/myapp/ap1/log/app_log.20131113というファイルを消すことになります。rmコマンドには-**iオプション**を付けて、削除前に確認するようにしています。

このようにファイルを削除するスクリプトでは、思わぬ動きをして誤ったファイルを消さないように注意を払う必要があります。ファイル削除など危険な操作を含むスクリプトを書く際には、まず次のように組み立てた文字列をechoして、想定したとおりのコマンドになっているかを一度確認するようにしましょう。

```
echo rm -i ${LOG_DIR}/app_log.$date_str
```

dialogコマンドのその他の機能

dialogコマンドには、この他にも次のようなさまざまな機能があります。詳しくは、man dialogしてオプションを調べてみてください。

dialogコマンドの主要なオプション

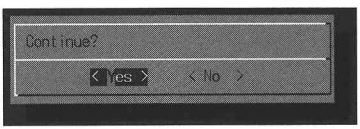
オプション	解説
--yesno	Yes/Noの問い合わせダイアログボックスを表示する
--msgbox	[OK]を表示するメッセージボックスを表示する
--inputbox	値を入力するインプットボックスを表示する
--fselect	ファイル選択画面
--dselect	ディレクトリ選択画面
--checklist	リストから複数をチェックできるチェックリストボックスを表示する
--radiolist	リストから1つだけ候補を選ぶラジオボタンリストを表示する
--menu	メニューリストを表示する

ここでは例として1つ、Yes/Noの問い合わせダイアログボックス(--yesno)の使い方を見てみましょう。リスト1がサンプルスクリプトです。

リスト1 問い合わせダイアログボックスの例

```
#!/bin/sh

dialog --yesno "Continue?" 5 40
answer=$?

if [ $answer -eq 0 ]; then
  echo "Selected: Yes"
elif [ $answer -eq 1 ]; then
  echo "Selected: No"
fi
```

Yes/Noの問い合わせダイアログボックス



Yes/Noの問い合わせダイアログボックスは、まずdialogコマンドを**--yesnoオプション**で起動します。1つ目の引数が表示するテキストで、普通は質問文となるでしょう。ここでは"Continue?"(続けますか?)と表示しています。次の数字はカレンダーと同様、ダイアログボックスの縦と横の大きさです。

スクリプトの中でYesが押されたかNoが押されたかを判断するには、dialogコマンドの**終了ステータス**を利用します。コマンドの終了ステータスは、シェルの特殊変数**$?**に保存されますが、dialogコマンドの場合は、Yesが選択されれば0が、Noが選択されれば1が終了ステータスとして返されます。

そこでこのスクリプトではシェル変数answerに$?を代入し、続くif文でYesの場合(終了ステータスが0の場合)とNoの場合(終了ステータスが1の場合)でそれぞれの答えを表示しています。

関連項目

031 作業ファイルディレクトリから、1年以上更新のないファイルを削除する
053 現在の前月を取得して、前月に作成されたログファイルを一括アーカイブする

No. 010 ファイルの圧縮中に、実行状態を示すプログレスバーを表示する

利用コマンド
pv, tar, gzip

キーワード
圧縮, プログレスバー

いつ使うか
長い時間がかかるファイル処理などを行う場合に、画面に進行状態を示したいとき

実行例

```
$ ./tar-pv.sh
 693MB 0:00:42 [16.4MB/s] [                    <=>       ]
```

スクリプト

```
#!/bin/sh

DATA_DIR=/myapp/datadir

cd $DATA_DIR ―――――――――――――――――――――――――――――――――――――――――――――❶
tar cf - bigfile1.dat bigfile2.dat | pv | gzip > archive.tar.gz ――❷
```

解説

このスクリプトは、tarコマンドとgzipコマンドで大きなファイルをアーカイブするときに、処理の進行状態をpvコマンドで表示するものです。

tarコマンドで現在処理中のファイルを表示するには、次のように**vオプション**を利用します。ここで、**cオプション**はアーカイブの作成(Create)、**fオプション**はアーカイブをファイルとして作成(ここではarchive.tar)することを意味しています。

❶tarコマンドで処理中のファイルを表示

```
$ tar cvf archive.tar *.dat
datadir/1001.dat
datadir/1002.dat
datadir/1003.dat
datadir/1004.dat
...
```

上記の出力結果を見るとわかるように、tarコマンドのvオプションはファイルごとの表示となるため、サイズの大きなファイルのアーカイブに時間がかかる場合などには、現在処理が進んでいるのか止まっているのか、判別がつきません。こんなときに役立つのが

pvコマンドです。

pvコマンドはPipe Viewerの略で、その名のとおりにパイプ処理中のデータの流れを可視化することができるコマンドです。このコマンドのインストール方法は、APPENDIXのP.400を参照してください。pvコマンドで利用される主なオプションを、次にあげてあります。

pvコマンドの主なオプション

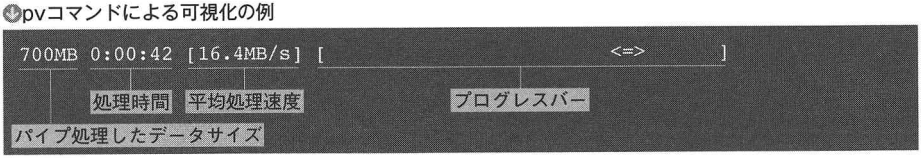
オプション	意味
-a	プログレスバーではなく、パイプを流れるデータの平均速度を表示する
-b	処理したバイト数のみを表示し、プログレスバーを出さない
-L	パイプの流量制限を行う。秒あたりの転送量をk (キロ)、m (メガ)、g (ギガ) 単位のバイト数で指定する
-q	何も表示しない静かな (quiet) モード。-Lで流量制限だけしたいときに用いる
-s	パイプを流れるデータサイズを先に指定する。これにより100%のプログレスバーを表示する

サンプルでは、まず❶でシェル変数DATA_DIRで指定されたディレクトリに移動しています。このディレクトリ内には、とてもサイズの大きいデータファイルが2つ、bigfile1.datとbigfile2.datとして保存されているものとします。続いて❷で、ファイルをまとめてtar.gz形式にアーカイブしています。tarコマンドの引数にハイフン"-"が指定されていますが、これはコマンドの出力をファイルではなく標準出力に出すよう指定しているものです。

これにより、tarアーカイブされたデータ列はパイプでpvコマンドにそのまま渡されます。pvコマンドは処理状態を標準エラー出力に表示しながら、同時にパイプの接続先であるgzipコマンドにデータ列を渡しています。**gzipコマンド**は、標準入力のtarアーカイブを受け取ってgzip圧縮し、archive.tar.gzというファイルに出力しています。

pvコマンドはデフォルトでは次のような表示を行います。この他にも流量制限(-L)などさまざまなオプションがありますので、詳しくはman pvとしてマニュアルを読んでみてください。

pvコマンドによる可視化の例



このようにpvコマンドは、標準入力をそのまま出力しながら、処理結果を標準エラー出力へ逐次わかりやすく表示する機能を持ちます。「データの流れを可視化するcatコマンド」と考えるとわかりやすいでしょう。

注意事項

・pvコマンドを用いて流量制限を行うには、次のように**-Lオプション**を利用します。例えばddコマンドを用いてout.datという1GBのファイルを作る際に、ほかのプロセスに影響を与えないように処理速度を制限するには以下のように記述します。

```
dd if=/dev/zero count=1024 bs=1024000 | pv -L 10m -s 1g > out.dat
```

pvコマンドのオプションに-L 10mと指定していますから、ddコマンドの結果は1秒間に最大10MBまでしかout.datファイルに出力されません。また、-s 1gとすることで総処理サイズは1GBであると明示的に指定しています。こうすると、プログレスバーが100%までの到達度で表示されるようになります。

関連項目

130 パイプ処理で各コマンドの終了ステータスを調べる

CHAPTER

02

変換処理

変数の取り扱いは、多くのプログラミング言語において重要なプログラムの要素となっています。シェルスクリプトでも、様々な設定値やファイル名・ディレクトリ名の操作を変数処理でおこないます。
この章では、変数定義の扱いやデフォルト値の設定、入力値の確認、シェル変数の記述などについて解説します。値が整数かどうかのチェックや、シングルクォートの扱いなど、シェルスクリプトでありがちな事例を取り上げて紹介します。

No. 011 実行時に変数の値が空のときは、デフォルト定義した値を設定する

利用コマンド
cp, tar

キーワード
未定義, デフォルト値

いつ使うか
環境変数が設定されている場合はその値を使い、設定されていなければスクリプトで決めたデフォルトの値を利用したいとき

実行例

```
$ ./var-parameter.sh
Extract files to /var/tmp.
```

スクリプト

```
#!/bin/sh

cp largefile.tar.gz ${TMPDIR:=/tmp} ―――――――――――――――❶
cd $TMPDIR
tar xzf largefile.tar.gz

echo "Extract files to $TMPDIR."
```

解説

このスクリプトは変数TMPDIRの値をチェックし、設定されている場合はその値を、設定されていないもしくは空文字列の場合は/tmpを作業ディレクトリとして、アーカイブファイルを展開するものです。

❶で書いている**:=**が、変数に値が設定されているかをチェックして値を代入する記法です。**${変数名:=値}**と書くことで、変数が設定されていないもしくは空のときは、指定した値を代入します。

つまり、環境変数TMPDIRに/var/tmpという値が事前に設定されていれば、このスクリプトは作業ディレクトリとして/var/tmpを使用します。一方、環境変数TMPDIRに値が設定されていないもしくは空のときは、スクリプトでデフォルト値と定めた/tmpを作業ディレクトリとして用いるということになります。

❶ではコマンドの一部として代入を行っていますが、スクリプトの先頭でまずこの処理を行って変数を初期化したいときは、次のように**:（ヌルコマンド）**を用いるとよいでしょう。

```
: ${TMPDIR:=/tmp}
```

このサンプル例のように、ある変数がすでに設定されているかどうかをチェックし、設定されていない場合はデフォルトの値を設定するというのは、環境変数を利用するシェルスクリプトでよく使われる方法です。

例えばユーザが、/large/tmpという巨大なディスクをマウントしたディレクトリを、作業ディレクトリとして使いたいとします。この場合、次のようにして環境変数TMPDIRを設定してからスクリプトを起動することで、スクリプト自体を修正せずに作業ディレクトリを指定することができるのです。

使用するときは起動前に環境変数をセットする

```
$ TMPDIR=/large/tmp; export TMPDIR
$ ./var-parameter.sh
Extract files to /large/tmp.
```

:-と:?と:+

この例の:=を用いてデフォルト値を設定すると、元の変数の値をデフォルト値で上書きしてしまいます。そうではなく、元の変数の値がセットされているかどうかだけをチェックして、値は上書きしたくない場合は:-を用います。

リスト1 一時的に変数をチェックしたい場合

```
#!/bin/sh

cp largefile.tar.gz ${TMPDIR:-/tmp} ―――――――――――❶
# この$TMPDIRは元の値がNULLならばNULLのままとなる
cd $TMPDIR ―――――――――――――――――――――――――❷
```

上記の例でTMPDIRが設定されていない場合、❶ではファイルは/tmpにコピーされます。しかし値の代入は行われないため、❷では引数なしのcdコマンドとなり自分のホームディレクトリ直下に移動してしまいます。

この他、似たような記法として:?と:+がありますので、以下に簡単に紹介しておきましょう。

```
${var:?message}
```

上記はシェル変数varの値を返します。ただし、シェル変数varが未定義もしくは空文字列のときは、messageを出力してスクリプトが終了します。これは次のように、変数が未定義もしくは空の場合はエラー終了するという用途に使われます。

```
# シェル変数MYDIRをチェック。未定義もしくは空ならエラー終了
: ${MYDIR:? シェル変数MYDIRがセットされていません}
```

次に:+の場合です。

```
${var:+word}
```

上記はwordを返します。ただし、シェル変数varが未定義もしくは空文字列のときは、nullを返します。これは次のように、「変数MYDIRが設定されている場合は1を返す」というようにif文のような記述ができます。

```
# シェル変数MYDIRをチェック。空文字列でない値ならばフラグflgに1をセット
flg=${MYDIR:+1}
```

関連項目

016 変数や関数を外部ファイルに記述する
021 未定義の変数をエラーとなるようにして、タイプミスを防ぐ

No. 012 関数の中でローカル変数を定義して、呼び出し元の変数を破壊しないようにする

利用コマンド
ls, local

キーワード
ローカル変数, グローバル変数, スコープ

いつ使うか
関数内で変数を扱う際に、呼び出し元に影響を与えないようローカル変数として定義したいとき

実行例

```
$ ./local-var.sh
directory: /home/user1/logdir
20131020.log  20131022.log

directory: /var/tmp
tmp.dat  tmp.3113
```

スクリプト

```
#!/bin/sh

DIR=/var/tmp

ls_home()
{
  # 変数DIRを関数内のローカル変数として定義
  local DIR ―――――――――――――――――❶

  DIR=~/$1 ―――――――――――――――――❷
  echo "directory: $DIR"
  ls $DIR
}

ls_home logdir ―――――――――――――――――❸

echo "directory: $DIR" ―――――――――――――❹
ls $DIR
```

解説

このスクリプトは、ホームディレクトリ直下のlogdirというディレクトリの中のファイルと、シェル変数DIRで指定された/var/tmpディレクトリの中のファイルを、**lsコマンド**で順に表示するだけの簡単なものです。

はじめにls_homeという関数を作っており、これはホームディレクトリ内の、引数($1)で受け取った名前のサブディレクトリを表示するものです。ここでは❸でlogdirという引数を与えていますので、❷での$1の値はlogdirとなります。

また、ホームディレクトリのパスを取得するため、❷で**~(チルダ)記法**を使っています。シェルスクリプトにおいて~(チルダ)はホームディレクトリのパスへと展開されます。つまり、この実行ユーザのユーザ名がuser1ならば、~/ は /home/user1/ となります。よって❷は実行時には、「DIR=/home/user1/logdir」となります。

さて、多くのプログラミング言語では、関数内で定義した変数は、その関数内のみ有効なローカル変数として扱われます。しかしシェルスクリプトでは、基本的に**変数はすべてグローバル変数**として扱われます。そのため関数内で変数の値を変更してしまうと、スクリプト全体に影響を及ぼすことになります。

つまり❷のシェル変数への代入文は、このスクリプトの最初で宣言しているDIR=/var/tmpに影響を与える恐れがあるのです。

これを防ぐため、このスクリプト例では、❶で**localコマンド**を使って変数を宣言しています。呼び出しの時点ですでにDIRというシェル変数が使われていますが、ls_home関数内の同名のシェル変数DIRはローカル変数なので、呼び出し元に影響を与えません。

もし関数内でlocalコマンドを使わないと、変数値が書き換えられてしまい、❹では/var/tmpではなく~/logdirが出力されてしまいます。

localコマンドを使わないと関数内で変数値が書き換えられてしまう

```
$ ./012-local.sh
directory: /home/user1/logdir
20131020.log  20131022.log

directory: /home/user1/logdir
20131020.log  20131022.log
```

関連項目

016 変数や関数を外部ファイルに記述する

No. 013 読み込んだHTMLファイルから特定の属性値を取得する

利用コマンド
expr

キーワード
パターンマッチ, 文字列, HTML, 属性

いつ使うか
変数内の文字列から、特定のパターンにマッチする部分を取り出したいとき

実行例

```
$ ./expr-match.sh
/about/
/sitemap/
/plan1.html
http://www.example.org/
```

スクリプト

```
#!/bin/sh

quote="[¥"']" ―――――❶
match="[^¥"']*" ―――――❷

while read line
do
  href=$(expr "$line" : ".*href=${quote}¥(${match}¥)${quote}.*") ―――――❸
  if [ $? -eq 0 ]; then
    echo $href
  fi
done < index.html
```

解説

このスクリプトは、カレントディレクトリにあるファイルindex.htmlからHTMLタグ中のhref属性を取り出し、その属性値を表示するものです。

HTMLファイルからある属性値を取得するには、**パターンマッチ**を利用して該当部分を取り出すのが簡易なやり方です。シェルスクリプトでパターンマッチするにはいろいろな手法がありますが、ここでは**exprコマンド**を使った例を見てみましょう。

exprコマンドは足し算などの数値演算をする目的でよく使われますが、「**expr 変数名：パターン**」の形で、正規表現を利用して変数内の文字列からマッチする部分を取り出すこともできます。この形では、パターン内でカッコ()でくくった部分を出力できます。なお、**カッコはエスケープする**必要があるため、実際の表記は「¥(パターン¥)」になります。例えば次の例では、"pen"が出力されます。

```
string="This is a pen."
expr "$string" : "This is a ¥(.*¥)."
```

サンプルでは、「href属性はダブルクォートもしくはシングルクォートで始まり、再びダブルクォートもしくはシングルクォートで終わる」というHTMLの書き方を考えて、まず❶でクォート部分を表現する正規表現を用意しました。クォート記号にはエスケープが必要で、実際のマッチ部分にこれを書くと大変煩雑になり読みづらくなるため、こうして別途シェル変数にしています。

続いて❷で、href属性の部分にマッチする正規表現を用意しました。これはクォート記号の中の文字列なので、「ダブルクォートもしくはシングルクォート以外の文字列」ということになるため、^を付けた否定の後ろにこれらクォート記号を並べています。

❸で実際にexprコマンドでファイルから読み込んだ1行1行をマッチしています。exprはマッチしたときのみ終了ステータスに0を返すので、**特殊変数$?**が0のときのみ表示することで、href属性値を出力できます。

注意事項

- exprコマンドは「重い」コマンドのため、速度はあまり速くありません。大量のテキスト処理には、sedやawkが向いています。
- 古いHTMLの書き方では、<a href=doc.html>のように属性値にクォートを付けずに裸で書いているものがあります。また、<a href="doc.html'>のようにダブルクォートで開いてシングルクォートで閉じているタイプミスもあるかもしれません。しかしここではクォートの扱い方の説明のため、そのような特殊な例は除外しました。

関連項目

018 HTMLファイルから、タグの中に書かれたコマンドを抜き出してそのまま実行する
029 複数HTMLファイルからtitleタグ部分のみを抜き出して、それぞれ別ファイルへ出力する
128 変数内の文字列の一部を置換する

No. 014 値が整数であることをチェックしてから計算を行う

利用コマンド
test, expr

キーワード
数値チェック、引数チェック、エラー処理

いつ使うか
exprコマンドなどで計算を行う前に、変数の値が整数かどうかのチェックを行いたいとき

実行例

```
$ ./int-check.sh 100a
Argument is not Integer.
$ ./int-check.sh 100
Argument is Integer.
110
```

スクリプト

```
#!/bin/sh

# 引数が整数かどうかをテスト
test "$1" -eq 0 2>/dev/null ―――――❶

if [ $? -lt 2 ]; then ―――――――――――❷
  echo "Argument is Integer."
  expr 10 + $1
else
  echo "Argument is not Integer."
  exit 1
fi
```

解説

このスクリプトは、コマンドライン引数で指定された整数値に10を足した値を返すものです。整数以外の値が指定された場合は、"Argument is not Integer."と表示してエラーになります。

足し算引き算などの**四則演算**は、**exprコマンド**で行うことができます。この際、かけ算の場合だけは*（アスタリスク）がシェルに解釈されないように、¥記号でエスケープする必要があることに注意してください。

```
expr $i + $j    # 足し算
expr $i - $j    # 引き算
```

```
expr $i ¥* $j   # かけ算
expr $i / $j    # 割り算
```

exprコマンドは小数に対応しておらず、**整数ではない値を四則演算しようとするとエラー**になります。そのためコマンド実行前に変数が正しい値かどうかをチェックしたい場合があります。特にこのサンプル例のように、ユーザからの入力で処理を行う場合は、きちんとエラー処理をしないと予期せぬ結果を招きます。

そのためこのスクリプトでは、❶で引数チェックを行っています。**testコマンド**を用いて、コマンドライン引数($1)が0と等しいかどうかを**-eq演算子**で確認します。testコマンドの出力結果自体は不要なので、標準エラー出力は/dev/nullへリダイレクトして表示しないようにしています。

❶の終了ステータス($?)は、次のようになります。

- コマンドライン引数が0と等しければ0
- コマンドライン引数が0と違う数値ならば1
- コマンドライン引数が0と比較できない文字列などならば2

そのため❷でこの終了ステータスを比較し、2より小さければ整数とみなしてそのまま計算を行い、そうでなければ整数ではないと判断してエラー終了としています。

注意事項

- このスクリプトでの整数値のチェックとして、文字列の"0000"などは、exprコマンドで整数の0として正常に計算できるためエラーとはしていません。

関連項目

007 選択式メニューを表示して、入力された数値の処理を実行する
101 IDカラムに"00001"などゼロ詰めで書かれたCSVファイルから、番号を指定して値を抽出する

No. 015 シングルクォートの中でシングルクォートを使う

利用コマンド
echo

キーワード
文字列, ダブルクォート, シングルクォート

いつ使うか
シングルクォートでくくった文字列中で、変数展開やシングルクォート記号を使いたいとき

実行例

```
$ ./single-quote.sh
It costs $100? I can't believe it!
```

スクリプト

```
#!/bin/sh

price=100
str='It costs $'$price'? I can'¥''t believe it!' ―――――――――――――❶
echo $str
```

解説

このスクリプトは、シェル変数strの中身を**echoコマンド**で表示するものです。変数strへの代入の際、文字列全体をシングルクォートでくくっていますが、そこでシェル変数priceの展開やシングルクォート記号そのものを扱うサンプル例です。

シェルスクリプトで文字列を扱うには、ダブルクォート(")とシングルクォート(')が使われます。基本的な文法ですがいろいろとややこしいことも多いので、以下、基本的なことを確認しながら解説します。

ダブルクォートでくくった文字列は、変数の展開とコマンド置換を行うため、$と`(バッククォート)はクォート内でも変数展開とコマンド置換の意味を持ったままとなります。一方、**シングルクォート記号は単純なクォート**で、変数展開などを一切行わず、'(シングルクォート)以外の記号をすべてそのまま出力します。

シングルクォートだと展開・置換を行わない

```
$ echo "My Terminal: $TERM"
My Terminal: xterm
$ echo 'My Terminal: $TERM'
My Terminal: $TERM
```

さて、ダブルクォートの中でダブルクォートを使うには、次のように¥記号でダブルクォートをエスケープすればうまくいきます。

```
str="He said ¥"Hello!¥". I said ¥"Hello¥"."
echo $str
```

しかしダブルクォートが多数使われる文字列では、いちいちエスケープするのが大変です。このような場合は、次のようにシングルクォートを用いて処理すると便利です。

```
str='He said "Hello!". I said "Hello".'
echo $str
```

しかしこれでは、シングルクォートの中なので変数展開が行われません。「ダブルクォートが多数登場するためシングルクォートで文字列を扱いたいが、そこで変数展開したりシングルクォートを使いたい」というための工夫がこのサンプル例になります。

サンプル例では、まずシングルクォートを利用しながら変数展開するための手法として、❶で'It costs $'でいったんシングルクォートを打ち切っています。その後の$priceはシングルクォートの外なので変数展開されます。その直後にまたシングルクォートを開いて、文字列を連結させています。

また、シングルクォートそのものを出力させるため、❶で'¥"という表記を使っています。これは、まずシングルクォートをいったん閉じ、「¥」でシングルクォート記号そのものを表示し、再びシングルクォートを開く、という意味です。

このように書けば、シングルクォートの中でも、変数展開やシングルクォート記号を扱うことができます。

関連項目

013 読み込んだHTMLファイルから特定の属性値を取得する
017 文章などの空白文字を含む文字列変数を引数にとるには

No. 016 変数や関数を外部ファイルに記述する

利用コマンド
.（ドットコマンド）

キーワード
外部ファイル，読み込み，定義ファイル

いつ使うか
複数のシェルスクリプトで共通の設定値や関数を使う際、それらの値や関数を外部ファイルとして定義したいとき

実行例

```
$ ./source.sh
09時28分38秒
large-file.tar.gz -> /var/tmp/myapp/large-file.tar.gz
09時28分55秒
```

スクリプト

```
#!/bin/sh

. ./env.sh ―――――――――――――――――――❶

nowtime
cp -i -v large-file.tar.gz "$WORK_DIR"
nowtime
```

解説

このスクリプトは、指定された作業ディレクトリに大きなファイルをコピーして、その処理時間を測る簡単なスクリプト例です。ここではlarge-file.tar.gzというファイルサイズの大きなファイルがあると仮定しています。

❶では、env.shという定義ファイルを.（ドットコマンド）で読み込んでいます。env.shの中身はリスト1のようになっており、テンポラリディレクトリを指定する変数（$WORK_DIR）と、現在時間をdateコマンドで表示する関数が定義されています。

リスト1 ファイルenv.shの中身

```
WORK_DIR=/var/tmp/myapp/

nowtime() {
  date +%X
}
```

このように、変数や関数を定義するとき、同一の定義を別のスクリプトでも共通して利用したい場合があります。その場合に1つ1つのスクリプトに定義を記載していくのは面倒ですし、後々に値の変更があった場合には、すべてのスクリプトを修正しなければいけません。これは手間がかかりますし、修正漏れなどの運用ミスも起こります。

これを防ぐためにも、変数や関数定義は共通の外部ファイルに設定して、スクリプトからはその外部ファイルを読み込む形式にしたい場合があります。このようなときに使うのが、このスクリプト例であげた.(ドットコマンド)です。

.(ドットコマンド)で外部ファイルを読み込んだ場合は、**あたかもソースファイルがそこにそのまま挿入されたかのようにファイル内のコマンドが実行**されます。つまり、❶の行は、リスト1の中身をそのまま該当箇所にエディタで貼り付けたかのように動きます。そのため外部ファイルで定義した変数や関数を取り込むことができるのです。

なお、ここで❶のところで「sh env.sh」とシェルスクリプトを「実行」した場合は、現在シェルとは別プロセスとして動作するため、変数は引き継がれません。以下が実行例ですが、このスクリプトを実行しても、❶は別プロセスとして実行されるので、変数や関数の定義が元のスクリプトに反映されません。結果として、次のようにエラーとなります。

変数や関数の定義が引き継がれないためエラーになる

```
$ ./source.sh
./source.sh: line 5: nowtime: command not found
cp: ./large-file.tar.gz and large-file.tar.gz are identical (not copied).
./source.sh: line 7: nowtime: command not found
```

.(ドットコマンド)による読み込みと、別プロセスとしての起動の違いはきちんと理解しておきましょう。env.shファイルには最初の行の#!/bin/sh (シバン)がないのも、単独で実行するからではなく、あくまで中身が別のシェルスクリプトに読み込まれて実行されることを想定しているからです。

なお、bashならば次のように.(ドットコマンド)と同一の動作である**sourceコマンド**が使えます。ドット1つでは見にくいため、bash環境でしか動かなくてかまわない場合は、sourceコマンドを使ったほうが見やすい書き方になります。

```
source env.sh
```

.(ドットコマンド)の功罪

外部ファイルを使うと、依存関係が生じます。つまり、あるシェルスクリプトをよその環境に移植する際、読み込んでいる外部ファイルも一緒に移動しなくてはいけません。

一方、シェルスクリプトのよさの1つとして、移植の際はスクリプトファイル1つをコピーするだけでよいという、お手軽さがあげられます。.(ドットコマンド)の利用で発生する依存関係は、この手軽さを損ないます。

そのため、このような依存関係が生じることを嫌って、.(ドットコマンド)は利用しないポリシーの人もいます。特にチームでの仕事の場合、あるいは将来的に長く使われるスクリプトでは、このような決めごとはチーム内ポリシーとして事前に策定しておくべきです。

注意事項

・ドットコマンドは、対象のファイルが存在しない場合はエラーになります。そのため読み込む設定ファイルは、事前にtestコマンドの-f演算子でファイルの存在チェックをしておくべきです。例えば以下は、Linux(CentOS)のsshdの起動スクリプトの一部です。/etc/sysconfig/sshdという設定ファイルが存在することをチェックしてから、&&(AND演算子)を用いてドットコマンドで設定ファイルを読み込んでいることがわかります。

```
[ -f /etc/sysconfig/sshd ] && . /etc/sysconfig/sshd
```

ドットコマンドを使う際、この書き方はよく使われますから覚えておくとよいでしょう。

関連項目

012 関数の中でローカル変数を定義して、呼び出し元の変数を破壊しないようにする

No. 017 文章などの空白文字を含む文字列変数を引数にとるには

利用コマンド
echo

キーワード
空白文字, スペース, 文字列, ダブルクォート

いつ使うか
空白文字（スペース）を含む文字列を、複数の文字列として扱われないようにしたいとき

実行例

```
$ ./space-str.sh
ERROR: invalid value
```

スクリプト

```
#!/bin/sh

result="invalid value"

if [ "$result" = "invalid value" ]; then ―――――――――――――――❶
  echo "ERROR: $result"  1>&2
  exit 1
fi
```

解説

このスクリプトは、空白文字を含む文字列をif文で比較し、値が"invalid value"という文字列だった場合はエラーを出力するスクリプト例です。空白を含む文字列の扱いがポイントです。

シェルスクリプトでは、変数の区切り文字はシェル変数**IFS**に定義されており、デフォルトでは空白記号・タブ・改行が指定されています。つまり空白文字は変数の区切りを表す特別な意味を持ちます。

そのため値として文字列を持つ変数が、その値に空白文字を含むときは、適切にクォートしなければ複数の変数として扱われ予期せぬ動作となることがあります。

❶では、変数resultが空白文字を含むため、これをクォートして"$result"としています。もしこのクォートを行わずに単に$resultと書くと、❶は次のように、イコール記号の左側に引数が2つあるものと解釈されてしまいます。

```
if [ invalid value = "invalid value" ]; then
```

この結果、引数が多すぎるというエラーになります。

クォートを行わない実行時エラー

```
$ ./space-str.sh
./space-str.sh: line 5: [: too many arguments
```

これを防ぐために❶では、空白文字を含む変数resultをダブルクォート記号でくくって"$result"と書き、空白文字を含めて全体を1つの文字列という扱いにしています。

また、もし変数resultの値が空っぽの場合は、❶は次のように解釈されます。

```
if [ = "invalid value" ]; then
```

これは比較対象がない状態でイコール記号を使ったことになり文法エラーとなります(なお、表示されるエラーメッセージはOS環境などにより多少変わります)。

resultが空のときの実行時エラー

```
$ ./space-str.sh
./space-str.sh: line 5: [: =: unary operator expected
```

シェルスクリプトで変数に文字列を入れる場合は、その値を利用する際に、**そこには空白文字が入っているかもしれない、あるいは空っぽの文字列かもしれない、ということを常に考慮**しなくてはいけません。基本的にはこのサンプル例のように、空白文字を含むにせよ含まないにせよクォートするのがよいでしょう。

関連項目

015 シングルクォートの中でシングルクォートを使う

No. 018 HTMLファイルから、タグの中に書かれたコマンドを抜き出してそのまま実行する

利用コマンド
sed, eval

キーワード
コマンド, 変数展開

いつ使うか
ファイルに書かれた文字列を抜き出し、コマンドとして実行したい場合

実行例

```
$ ./eval.sh
Sun Nov 10 15:05:34 JST 2013
-rw-rw-r--. 1 user1 user1 11968 Oct 26 12:32 myapp.log
```

スクリプト

```
#!/bin/sh

filename="myapp.log"
eval $(sed -n "s/<code>¥(.*¥)<¥/code>/¥1/p" command.htm)
```

解説

このスクリプトは、command.htmファイル内に書かれた<code>タグを抜き出し、その要素をコマンドとして実行するものです。

ここで用意するcommand.htmファイルの中身は、リスト1のようになっています。つまり、「date; ls -l $filename」をコマンドとして実行することを想定しています。この際、シェルスクリプト内で$filenameの変数展開も行います。

リスト1 処理対象のHTMLファイル例(command.htm)

```
<html>
<head><title>Code List</title></head>

<body>
<p>This is a sample code.</p>
<code>date; ls -l $filename</code>

</body>
</html>
```

このスクリプトでは、まず<code>タグの部分をパターンマッチで取り出します。シェルスクリプトでマッチ部分の文字列を取り出すにはさまざまな手法がありますが、ここでは次のように**sedコマンド**を用いています。

```
sed -n "s/<code>¥(.*¥)<¥/code>/¥1/p" command.htm
```

sedコマンドの**-nオプション**は、処理後にパターンスペースの内容を出力しないようにするオプションです。そのままでは何も出力されずに意味がないので、最後にpフラグを付けて、マッチした場合のみパターンスペースを出力するように指定しています。

このように-nオプションとpフラグを組み合わせることで、sedコマンドで置換を行う際に「**置換が発生した行だけ出力する**」ことができます。これはsedコマンドでよく使われる手法ですので覚えておきましょう。

さて今回はマッチした行のうち<code>タグでくくられた部分のみ取り出したいので、上記では**後方参照¥1**を使っています。sedコマンドでのマッチの際、カッコ()でくくった部分は前から順番に¥1, ¥2,...と参照することができます。ここでは<code>内の任意の文字列.*をカッコ()でくくって、¥1で取り出せるようにしています。この¥1の部分が実行したいコマンド文字列となります。

なお、sedの正規表現では後方参照する部分のカッコとして()ではなく¥(¥)を用いるため、この例でも¥(¥)でくくっています。

これでコマンド文字列が取り出せたので、**eval**により変数展開を行ってコマンドを実行します。evalは引数に与えられた文字列を変数展開してから、コマンドとして実行します。ここではsedコマンドの出力は、次のような文字列となっています。

```
date; ls -l $filename
```

evalコマンドにこの文字列を引数として与えると、シェル変数filenameが置き換えられ、結果として、次のようなコードが実行されることになります。

```
date; ls -l myapp.log
```

このようにevalコマンドを用いると、シェルスクリプトのコード自体を動的に生成して実行することができるようになります。

なおevalコマンドは、使い方によってはメタプログラミングのような書き方ができるとても便利なコマンドです。しかし、ただのテキスト文字列をコマンドとして実行するため、使い方によっては悪意のある人がプログラムを注入することができ、とても危険な結果をもたらす可能性もあります。

OSコマンドインジェクションなどの脆弱性をもたらす原因にもなるため、ユーザが自由に入力できる文字列など外部からの入力値をそのままevalする、という使い方は決してしないように注意してください。

注意事項

・このサンプルではsedで<code>タグを処理しているため、次のように<code>タグ内で改行を入れていると正しく動作しません。

```
<code>
 date; ls -l $filename
</code>
```

・evalコマンドは引数の文字列をコマンドと解釈しそのまま実行してしまうため、入力値には注意しなくてはいけません。例えば、もしこのサンプル例で次のような文字列が<code>タグに書かれていたとします。

```
rm -rf ~/*
```

この場合、スクリプトを実行するとチルダ(~)がホームディレクトリとして展開されるため、実行者のホームディレクトリ内のすべてのファイルが消去されてしまいます。
evalを実行するスクリプトを書く際は、まず**evalを記述している部分をechoに置き換えて**みて、実行される実際のコマンド内容をよく確認しましょう。またユーザの入力値など、何が入ってくるかわからない文字列をevalするのも危険です。

関連項目

013 読み込んだHTMLファイルから特定の属性値を取得する

No. 019 アンダースコアなどを含む文字列内で、変数の区切りを明示的にする

利用コマンド
wc

キーワード
変数名, 文字列, 区切り

いつ使うか
変数名の後ろに文字列を連結する際に、変数名の区切りを明示したいとき

実行例

```
$ ./varname.sh
342 20131106_log
```

スクリプト

```
#!/bin/sh

today="20131106"
# シェル変数 today を正しく展開する
wc -l ${today}_log
```

解説

このスクリプトは、20131106_logというファイル名を持つログファイルの行数を表示するものです。**wcコマンド**はファイルの文字数や行数を調べるコマンドで、**-lオプション**を利用すると、ファイルの行数が表示されます。ここではファイル名が「シェル変数名＋アンダースコア＋log」と、シェル変数名の後ろにアンダースコアが来る場合の注意点を示しています。

シェルスクリプトの変数名として使える文字は、**アルファベット・数字・アンダースコア**の3種類です。ここで、あるシェル変数を使う際に、その変数の後ろに文字列を連結したいとします。この際、後ろに続く文字がシェルスクリプトの変数名として使える文字の場合には、シェルはできるだけ長い変数名を使おうとして後ろの文字までも変数名とみなしてしまいます。

例えば次は、catコマンド実行の際にシェル変数todayが"20131106"と展開されることを期待して20131106_logという名前のファイルを表示しようと意図したものです。

```
today=20131106
cat "$today_log"
```

しかしこのパターンでは、シェルは実際にはtoday_logという名前のシェル変数だと解釈してしまいます。このような変数は定義されていないので、結果として「cat ""」と空文字列に対してcatコマンドを実行することになり、実行結果はファイルが見つからずエラーとなります(おそらく、No such file or directoryと表示されることでしょう)。

このようなケースで、シェル変数の後ろに文字列を連結したい場合には、変数名の区切りを明確にする中カッコ{}を用います。つまり、${today}_logと書けば、todayがシェル変数だと明示的に指定できるため、後ろにシェル変数に使える文字が続いていても正しく扱うことができます。

なお、中かっこが必要ない場面でも、見やすくするためだけにシェル変数を中カッコでくくってもかまいません。コードが読みにくいと感じたら、文法上必要のない場合でも中カッコを付けることをお勧めします。

bashの配列変数

bash限定のトピックとなりますが、配列変数を扱う際には必ず変数名に{}を付けなくてはいけません。

例えば次の実行例では、配列変数numberに{}を付けた場合には、正しく"one"という値が取り出せています。

一方、{}を付けないと「$numberに文字列"[1]"を連結」という意味になってしまいます。この場合、配列変数は添え字を付けずに参照すると先頭の要素が参照できるため、先頭の要素"zero"に"[1]"を連結し、「zero[1]」という文字列が得られてしまいました。

配列変数には{}を付ける

```
$ declare -a number=("zero" "one" "two")
$ echo ${number[1]}
one                                    期待どおり配列を取得
$ echo $number[1]
zero[1]                                [1]が文字列として表示
```

関連項目

021 未定義の変数をエラーとなるようにして、タイプミスを防ぐ

No. 020 コマンドの出力結果を用いてファイル名を組み立て、そのファイル名を対象にコマンドを実行する際に見やすくする

利用コマンド
hostname, grep

キーワード
コマンド置換, 入れ子, ネスト

いつ使うか
コマンド置換の処理をネスト(入れ子)にしたいとき

実行例

```
$ ./comsub.sh
Error counts: 2
```

スクリプト

```
#!/bin/sh

err_count=$(grep -c "ERROR" /var/log/myapp/$(hostname).log)  ❶
echo "Error counts: $err_count"
```

解説

このスクリプトは、**hostnameコマンド**でファイル名を組み立てたログファイルの中から、ERRORという文字列を検索し、マッチした行数を表示するものです。検索と行数のカウントには、**grepコマンド**の**-cオプション**を利用しています。

この例では、**コマンド置換**を行う際に、通常使われる`(バッククォート記号)ではなく**$()**を使っていることがポイントです。

このスクリプトの前提条件として、検索対象のログファイルが以下のように設置されているものと仮定しています。

- /var/log/myappというディレクトリの中に、"(サーバ名).log"というファイル名のログが出力されている
- このスクリプトを実行するサーバのホスト名は"server1"である(hostnameコマンドを実行するとserver1と出力される)
- server1.logというテキストファイルの中身はリスト1のようになっている

リスト1 サンプルで使用するserver1.logの中身

```
2013/11/13 21:10:22 [INFO] script start.
2013/11/13 21:10:24 [ERROR] File does not exist: /var/tmp/foo.txt
```

```
2013/11/13 21:10:24 [ERROR] File does not exist: /var/tmp/bar.txt
2013/11/13 21:10:25 [INFO] script end.
```

一般的にシェルスクリプトで、あるコマンドの出力結果をそのままスクリプト中でシェル変数に代入して使いたい場合には、`が用いられます。これは**コマンド置換**(Command Substitution)と呼ばれる機能です。例えば次は、dateコマンドの出力を利用して今日の日付をYYYYMMDD形式でシェル変数date_strに代入する例です。

```
date_str=`date +"%Y%m%d"`
echo $date_str
```

しかしこの`を用いたコマンド置換は、処理をネスト(入れ子)にするときに手間がかかります。サンプルの❶を、`を用いた手法で書く場合は、次のように内側の`(バッククォート記号)を¥でエスケープしないといけません。

```
err_count=`grep -c "ERROR" /var/log/myapp/¥`hostname¥`.log`
```

このようにいちいちエスケープするのは大変ですし、既存のスクリプトを修正して入れ子にする変更を加える場合には、エスケープ忘れによるバグを生み出す元ともなります。そこでお勧めするのが、このサンプルで用いている**$()**という記法です。

この記法が優れているのは、**処理をネストする際にも既存のコマンド置換部分をエスケープする必要がない**点です。このため、既存のコードを修正する際に処理内部のコードを修正する必要がないため、保守性に優れています。

また、カッコの対応というわかりやすい書き方でコマンド置換部分を記述できるのも、**コードが読みやすくなる**ため大きな利点です。

一般にviやemacsなど多くのエディタでは、プログラミングの助けとなるよう、カッコの対応がひと目でわかるような工夫がされています(例えばviでは、カッコの上にカーソルを置いて%キーを押すと対応するカッコに移動することができます)。そのため、シェルスクリプトでネストするコマンド置換を行う際は、`ではなく、この$()記法で書くことをお勧めします。

関連項目

033 ファイルをバックアップする際にファイル名に日時を入れる
059 ホスト名からIPアドレスを取得する
129 中間ファイルを作らずにコマンドの出力をファイルのように扱う

No. 021 未定義の変数をエラーとなるようにして、タイプミスを防ぐ

利用コマンド
set

キーワード
未定義, 変数, エラー , 空文字列

いつ使うか
スクリプト中で未定義の変数を利用した場合は、エラーとして終了するようにしたいとき

実行例

```
$ ./set-u.sh
./set-u.sh: line 7: COP_DIR: unbound variable
```

スクリプト

```
#!/bin/sh
set -u ―――――――――――――――――❶

COPY_DIR=/myapp/work

# COPY_DIR と間違えて COP_DIR と打ってしまった！
cp myapp.log $COP_DIR
```

解説

このスクリプトは、変数名をタイプミスして未定義の変数を利用してしまった際にエラーを表示するものです。実行例では、"unbound variable"とエラーになっていることがわかります。

通常、シェルスクリプトでは、**宣言されていない変数を利用してもエラーとはなりません。**変数への代入は変数自体を宣言せずにどこでもできますし、未定義の変数は参照しようとすれば空文字列となります。そのため、いちいち変数宣言をすることなく気軽にプログラミングできるのがシェルスクリプトの利点の1つです。

しかし、変数宣言が必要ないというシェルスクリプトのこの特徴は、思わぬバグを招く危険性があります。例えばrmコマンドで削除するパス名をシェル変数で指定する場合は、細心の注意を払わなくてはいけません。

この例として、リスト1を見てください。これは、シェル変数dirnameで指定された/myapp/work/tmpdirディレクトリを削除するスクリプト例です。

リスト1 シェル変数で指定されたディレクトリを削除する例

```
#!/bin/sh

dirname=/myapp/work/tmpdir
rm -rf $dirname/
```

しかしこのシェルスクリプトは、もし$dirnameの部分をミスタイプして$dirnamなどとしてしまった場合、シェル変数dirnamは未定義のため空文字列となり次が実行されます。

```
rm -rf /
```

これは/(ルートディレクトリ)を消してしまう操作で、システム全体の破壊を伴う、大変に危険なスクリプトです。このように、変数名をタイプミスしてそこが空文字列となってしまい、その結果思わぬファイルを削除してしまうというのはよくあるバグであり、致命的な結果をもたらす場合があります。

これを防ぐため、このサンプル例では❶でsetコマンドの-uオプションを用いています。**set -uを指定すると、スクリプト内で未定義の変数を参照しようとした時点でエラー**となり、シェルスクリプトの実行が停止します。結果として、未定義の変数を利用しようとしたコマンドの実行を防ぐことができます。

rmコマンドなど危険な動作を行うシェルスクリプトを書くときは、基本的にset -uして、変数名が未定義ならばエラーとするようにしたほうがよいでしょう。

set -uのデメリット

set -uする副作用として、コマンドライン引数$1などが扱いづらくなる点があげられます。次は、コマンドライン引数を表示するだけの簡単なスクリプトです。

リスト2 コマンドライン引数を表示するスクリプト例

```
#!/bin/sh

set -u

echo "1st arg: $1"
echo "2nd arg: $2"
```

ここで、もしset -uしていなければ指定されていないコマンドライン引数は参照時に空文字列となるため、実行結果は次のようになります。

引数を1つだけ指定した場合

```
$ ./arg-set.sh 1
1st arg: 1
2nd arg:
```

しかしこのスクリプトはset -uしているため、コマンドライン引数が1つしかない場合は$2が未定義となり、結果として次のようにエラーとなります。これを防ぐには、シェルスクリプト内でコマンドライン引数を自分で数えて処理を書かなければいけません。

set -uした場合の動作

```
$ ./arg-set.sh 1
1st arg: 1
./arg-set.sh: line 5: $2: unbound variable
```

シェルスクリプトではコマンドライン引数を扱うことが多いため、利便性のためset -uを付けずに書くことも多いです。

何でもかんでもset -uを付けておけばよい、というわけではないことに注意してください。

関連項目

011 実行時に変数の値が空のとき、デフォルト定義した値を設定する

019 アンダースコアなどを含む文字列内で、変数の区切りを明示的にする

No. 022 ヒアドキュメントで変数展開をせずにそのまま$strのように表示する

利用コマンド

cat

キーワード

ヒアドキュメント，クォート，パラメータ展開，コマンド置換，テキスト

いつ使うか

ヒアドキュメント本体に、$記号や`（バッククォート記号）を使う際に、展開せずにそのまま出力したいとき

実行例

```
$ ./here.sh
ここはヒアドキュメント本体です。
この部分に書かれた文字列は、コマンドの標準入力に
直接リダイレクトされます。

終了文字列をシングルクォート記号でクォートしているので、
$str など書いても変数展開されませんし、
`echo abc` としてもコマンド置換されません。
```

スクリプト

```
#!/bin/sh

# この変数は展開されないので実際には利用されない
str="Dummy"

cat << 'EOT'                                    ❶
ここはヒアドキュメント本体です。
この部分に書かれた文字列は、コマンドの標準入力に
直接リダイレクトされます。

終了文字列をシングルクォート記号でクォートしているので、
$str など書いても変数展開されませんし、
`echo abc` としてもコマンド置換されません。
EOT
```

解説

このスクリプトは、ヒアドキュメントを利用した際にパラメータ展開やコマンド置換が行われないようにするものです。

ヒアドキュメントとは、シェルスクリプト本体に埋め込んだテキストを、スクリプト内

のコマンドの標準入力として利用する機能のことです。ここにドキュメントがあるよ、という意味でヒアドキュメント(Here Documents)と呼ばれます。

一般的に、ヒアドキュメントを利用する場合は、次のように記述します。

書式 ヒアドキュメントの使い方

```
(コマンド) << (終了文字列)
ヒアドキュメント本体
…
終了文字列
```

終了文字列は、ヒアドキュメント本体に表れない文字列であれば何でもかまいません。慣用的にはEND, EOT, EOFなどがよく使われますが、誤ってヒアドキュメント本体にこの文字列が表れるとエラーを引き起こすため、__EOT__など記号を合わせて用いたり、EndOfMultilineTextなど長い終了文字列を指定する書き方を好む人もいます。ここではEnd Of Textという意味で、EOTを用いました。

さて、ヒアドキュメント部分では**パラメータ展開**と**コマンド置換**が行われます。つまり$記号と`記号は特別な意味を持ち、例えば変数$strは変数値へと展開されてしまいます。

一方、パラメータ展開やコマンド置換を行わず、書かれた内容一切をそのまま出力したい場合のスクリプト例が、本サンプルになります。ヒアドキュメントの終了文字列を❶のように**シングルクォートでくくって**'EOT'とすることで、ヒアドキュメント本体のパラメータ展開やコマンド置換を抑制し、$記号や`記号を含んだテキストをそのまま扱うことができます。

パラメータ展開の抑制

ヒアドキュメントの中で、パラメータ展開したい変数としたくない変数が混在する場合があります。このような場合は、個別に変数をエスケープすることで対応できます。

リスト1 エスケープするとパラメータ展開を抑制できる

```
#!/bin/sh

string="Hello"

cat << EOT
この変数は展開されます: $string
この変数は展開されません: ¥$string
EOT
```

上記では、1つ目のシェル変数$stringはエスケープしていないため変数展開されます。一方、2つ目は¥$stringと$記号をエスケープしているため、$stringという文字列そのものを意味します。結果として、このスクリプトの実行結果は次のようになります。

⬇パラメータ展開を抑制した実行例

```
この変数は展開されます： Hello
この変数は展開されません： $string
```

ヒアストリング

bashには、ヒアドキュメント(<<)とよく似た記法として**ヒアストリング**(**<<<**)が実装されています。これを用いると、より簡潔に埋め込みテキストをシェルスクリプトに記述することができます。

リスト2 ヒアストリングの例

```
#!/bin/bash

string="Hello"

# 埋め込み文字列はダブルクォートでくくるだけでよい
cat <<< "あいさつ文のサンプル：
こんにちは
$string
ニーハオ"
```

ヒアドキュメントではEOTなどの終了文字列を利用しましたが、ヒアストリングでは単にダブルクォートで埋め込みたい文字列をくくればよいだけなので、より直感的でわかりやすくなります。

なお、埋め込む文字列内の$記号を変数展開させたくない場合は、ヒアドキュメントと同様に、ダブルクォートではなくシングルクォートでくくるようにします。

関連項目

015 シングルクォートの中でシングルクォートを使う

020 コマンドの出力結果を用いてファイル名を組み立て、そのファイル名を対象にコマンドを実行する際に見やすくする

CHAPTER

03

ファイル処理

UNIXの運用管理において、バックアップのためのコピーや不要ファイルの削除など、ファイル操作を行う機会は数多くあります。このような処理は、スクリプトで自動化するとよいでしょう。
この章では、様々なファイル操作のサンプル例を紹介します。大量のログファイルへの一括処理やtarコマンドでの一部ファイルの除外、2つのディレクトリ内の比較など、運用時に何かと必要と思われる処理をまとめてあります。

No. 023 絶対パスで起動されても相対パスで起動されても、同じ動作をできるようにする

利用コマンド
cd, dirname

キーワード
絶対パス, 相対パス, フルパス, cron

いつ使うか
cronなどからスクリプトをフルパスで起動する際に、相対パスで起動した時と同じ動きをさせたいとき

実行例

```
$ cd /home/user1
$ /home/user1/myapp/dirname.sh
START
END
```

スクリプト

```
#!/bin/sh

cd "$(dirname "$0")" ―――――――――――――――――――――❶

./start.sh
./end.sh
```

解説

このスクリプトは、2つの外部スクリプトファイルstart.shとend.shを順に実行するものです。ここで、start.shとend.shの2つのファイルは/home/user1/myappディレクトリに設置されており、それぞれの中身は、単に"START"および"END"と表示するだけのスクリプトであるとします。

このようにシェルスクリプトの中で他のシェルスクリプトを実行する際には、パスの書き方に注意が必要です。あまり注意していないと、サンプルスクリプトは、リスト1のように書いてしまうでしょう。

リスト1 カレントディレクトリを意識せずに書いた例

```
#!/bin/sh

./start.sh
./end.sh
```

スクリプトを書いているときは、シェルスクリプトファイルが置いてあるディレクトリをカレントディレクトリとして作業することが多いため、リスト1のスクリプトは正常に動作します。しかし、スクリプトが完成して、cronに登録して定期稼働するように設定すると、このスクリプトは次のようなエラーとなってしまうかもしれません。

cron登録時の動作結果

```
/home/user1/myapp/dirname.sh: line 3: ./start.sh: No such file or
directory
/home/user1/myapp/dirname.sh: line 4: ./end.sh: No such file or directory
```

このように、「手で実行してみると正常だったが、cronに登録して定期バッチにすると動かなくなった」というのは非常にありがちな事例です。これは、cron起動の際には、カレントディレクトリが**cron実行ユーザのホームディレクトリ**となってしまうことが原因です。

つまりcronからdirname.shが実行されたときにはカレントディレクトリは/home/user1となっているため、ここで./start.shを指定すると、/home/user1/start.shというファイルが探されてしまうのです。

このようにスクリプト内で他のシェルスクリプトを実行するプログラムでは、リスト2のように外部のスクリプトファイルをフルパスで指定してしまうのも1つの手です。

リスト2 必ずフルパスで指定する方法もあるが移植性が低い

```
#!/bin/sh

/home/user1/myapp/start.sh
/home/user1/myapp/end.sh
```

しかし、その場合、このスクリプトを置いているディレクトリ(ここではmyapp)の名前を変えることができなくなりますし、よそのサーバへコピーする際にも移植性が失われます。そのため、相対パスで書きたいケースのほうが多くなります。

この問題を解決するには、シェルスクリプトがまずはじめに、「自分が置かれているディレクトリにcdコマンドで移動してから処理を開始する」という動きをすればよいことになります。この動きを実現しているのが、サンプルの❶です。

❶で使っている**dirnameコマンド**は、フルパスが与えられた際にディレクトリ部分を取り出すことができます。**$0**というのはシェルが実行された時のコマンド自身を表す変数で、この例では"/home/user1/myapp/dirname.sh"が入っています。

ディレクトリ部分を取り出すdirnameコマンド

```
$ dirname "/home/user1/myapp/dirname.sh"
/home/user1/myapp
```

一方、**$()**は**コマンド置換**の記法で、コマンドの出力をそのままスクリプト中で利用することができます。つまりこの❶は、「dirname "$0"の出力結果のディレクトリに、cdコマンドで移動する」という意味になります。結果として、この例では/home/user1/myappディレクトリにcdコマンドで移動します。

これにより、シェルスクリプトは自分が置かれているディレクトリにcdコマンドで移動できます。こうすれば、シェルスクリプトが相対パスで起動されても絶対パスで起動されても、外部のシェルスクリプトは相対パスで実行することができます。

注意事項

- ❶にて、$0およびコマンド置換全体をダブルクォーテーション記号でクォートしているのは、ディレクトリ名にスペース(空白文字)を含む場合にも正常に動作するようにするためです。
- dirnameコマンドを使わずに、次のような書き方をすることもあります。

```
cd "${0%/*}"
```

これはシェルの**パラメータ展開**を利用した書き方です。${parameter%word}と書くことで、変数parameterの値から、wordに後方一致でマッチする部分を削除した値を得ることができます。つまり上記の例では、wordとして/*が指定されているため、「変数$0の後ろ側から『/任意の文字列』を削除した値」、すなわちディレクトリの部分のみ取り出すことができます。上記の記法は、dirnameコマンドという外部コマンドを利用せずにシェルの機能だけで実現できるため、こちらの書き方を好む人もいます。知識として覚えておいたほうがよいでしょう。

関連項目

016 変数やファイルパスなどを外部ファイルに記述する
017 文章などの空白文字を含む文字列変数を引数にとるには

No. 024 コマンドの使い方を表示する際に、現在の自分自身のファイル名を使って例示する

利用コマンド
basename

キーワード
ファイル名, コマンドライン引数, スクリプト名

いつ使うか
ヘルプ表示やログ出力の際に、自分自身のファイル名を出力したいとき

実行例

```
$ /home/user1/myapp/basename.sh
Usage: basename.sh <string>
$ /home/user1/myapp/basename.sh HELLO
Start: basename.sh ...
 Input Argument: HELLO
Stop: basename.sh ...
```

スクリプト

```
#!/bin/sh

prog=$(basename "$0")                                    ❶

# 引数が1つではない場合は、ヘルプを表示して終了する
if [ $# -ne 1 ]; then                                    ❷
  echo "Usage: $prog <string>" 1>&2                      ❸
  exit 1
fi

# コマンドライン引数 $1 を表示
echo "Start: $prog ..."
echo " Input Argument: $1"                               ❹
echo "Stop: $prog ..."
```

解説

このスクリプトは、以下の条件で**コマンドライン引数**を表示するものです。

・コマンドライン引数が1つだけの場合はその値を表示する
・コマンドライン引数がない、もしくは2つ以上ある場合は、使い方が間違っているとしてヘルプを表示する

❶で自分自身のファイル名をスクリプト内で取得しており、その値をシェル変数progに格納してヘルプ表示などに使っていることが、このサンプルのポイントです。

シェルスクリプトはその性質から移植性に優れたプログラムであるため、運用上、コピーしてよそで使われる場合も多々あります。この際、移植先ではファイル名を変更して使われるかもしれません。そのようなときにファイル名を決め打ちしてスクリプト内で扱っていると（そのような書き方を**ハードコード**と言います）、実際のファイル名と食い違うために混乱を招きます。

そのためこのスクリプトでは、❶で自分自身のファイル名を参照して取得しています。ここで使われている**basenameコマンド**とは、次のように、ファイルパスを示す文字列からパス部分を取り除いて、ファイル名のみを抽出するコマンドです。

basenameコマンドはファイル名のみを取得する

```
$ basename "/home/user1/myapp/script.sh"
script.sh
$ basename "./script.sh"
script.sh
```

❶では、変数$0に対するbasenameコマンドの出力を、**コマンド置換$()**によってシェル変数progに代入しています。ここで変数$0とはシェルスクリプトで用いることのできる特殊な変数で、このシェルスクリプトが起動されたときのコマンド名となります。このシェルスクリプトはフルパスで実行されたかもしれませんし、相対パスで実行されたかもしれませんが、上記のようにどちらであっても正しくファイル名を取得することができます。

なお、この特殊な変数$0と似たものに、コマンドライン引数を表す**位置パラメータ**（→P.4）がありますが、シェルでは**$0**は位置パラメータではなく**特殊パラメータ**として扱われています。詳しく知りたい方は、man shとしてマニュアルの特殊パラメータ（Special Parameters）の項を読んでみてください。

続いて❷で、コマンドライン引数のチェックをしています。変数**$#**には、コマンドに与えられた引数の数が入っているため、これが1でない場合は引数のチェックでエラーとしています。この際、エラー時のメッセージとして❸のようにUsage（使い方）を表示しています。これはエラーメッセージなので、1>&2と書くことで標準エラー出力に表示しています。

このようにUsageを表示する場合に、$0から取得したスクリプトのファイル名を使ってヘルプを表示することは、よく使われる手法ですから覚えておきましょう。例えば次のように、ファイル名をハードコードするのは好ましくありません。

```
echo "Usage: basename.sh <string>" 1>&2
```

いまはたまたまファイル名とこの記述が一致しているからよいかもしれませんが、誰か

がこのスクリプトを他のマシンに移植するためにコピーしてファイル名を変更したら、スクリプト内の記述も書き換えないといけません。これは保守性を落とす書き方です。

一方、自分自身のファイル名は$0から取得するようにしておけば、ファイル名を変更してもスクリプト本体の修正は必要ありません。後々のメンテナンス性も考えて、プログラムで使う値は、なるべくハードコードしないようにしましょう。

❹では、指定された引数の値を出力しています。$1とは、コマンドライン引数の1番目の値を指します。またここでは、スクリプトの動作ログとして変数progの値を合わせて出力することで、後でログファイルを見たときにどのスクリプトが出力したログなのかがわかりやすくなるようにしています。

注意事項

- 自分自身のファイル名を取得する場合、basenameコマンドの代わりに次のような書き方をされている場合があります。

```
prog=${0##*/}
```

これはシェルの**パラメータ展開**を利用した書き方です。**${parameter##word}**と書くことで、変数parameterの値から、wordに最長マッチする部分を削除した値を得ることができます。つまり上記の例では、wordとして*/が指定されていますから、「変数$0から『任意の文字列/』を削除した値」、すなわちファイル名のみを得ることができます。

- サンプル例のように、Usageのエラーメッセージを表示する際に$0で自分自身のスクリプト名を表示する手法は、システムツールなどにもよく使われます。例えばLinuxならば、/etc/init.d配下のシェルスクリプトに「grep '$0'」と検索してみると、この手法を用いている起動スクリプトが多く見つかります。参考に見てみるとよいでしょう。

関連項目

023 絶対パスで起動されても相対パスで起動されても、同じ動作をできるようにする
041 拡張された.htmと.htmlが混じったHTMLファイル群の拡張子を一括してtxtに変更する

No. 025 ディレクトリ移動した後に簡単に元の場所に戻る

利用コマンド
cd, echo, tar

キーワード
サブシェル, カレントディレクトリ, 子プロセス

いつ使うか
スクリプト内でcdコマンドにより別のディレクトリに移動した後、元のディレクトリへ簡単に戻りたいとき

実行例

```
$ ./subcd.sh
Archive: /var/tmp/archive.tar
count.txt
data1.txt
data2.txt
Start: command.sh
```

スクリプト

```
#!/bin/sh

# カッコ内はサブシェルとなるため、ディレクトリ移動はこの中だけに影響する
(
  echo "Archive: /var/tmp/archive.tar"
  cd /var/tmp
  tar cvf archive.tar *.txt
)

# スクリプト実行時のカレントディレクトリ内で処理
echo "Start: command.sh"
./command.sh
```

❶

解説

このスクリプトは、/var/tmpにカレントディレクトリを移動して、その中にある拡張子txtのファイルをarchive.tarというファイルにアーカイブしてから、元のディレクトリに戻ってきてcommand.shを実行するものです。

ここでcommand.shというのは、このサンプルスクリプトと同じディレクトリ内に設置された、何らかの処理を行うスクリプトであると仮定しています。

シェルスクリプト内では、作業用のディレクトリに移動するためなど、cdコマンドでカレントディレクトリを移動して処理を行いたいときがあります。この際、処理が終わっ

た後に元のディレクトリに戻ってくるには、最初に自分がいたディレクトリ名を保存しておかなくてはいけません。

しかし、もといたディレクトリを保存しておかなくても、簡易な方法で元のディレクトリに戻ってくることができます。それがこのスクリプトの例としてあげている、サブシェルを利用する方法です。❶がサブシェルの記法で、カッコ()でくくった部分がサブシェルとして実行されます。

サブシェルとは、現在のシェルの中で新しく起動されるシェルのことです。**サブシェル中では呼び出し元の環境は引き継がれますが、サブシェル内での環境の変更は、呼び出し元のシェルに影響を与えることはありません。**

すなわち、サブシェルの中でカレントディレクトリを変更したり、変数の値を変えたりしても、元のシェルのカレントディレクトリや変数の値は変わりません。このような関係を、UNIXでは「親と子」と呼びます。つまりサブシェルは子プロセスで、呼び出し元が親プロセスです。

このサンプル例では、サブシェルの「呼び出し元の環境を変えない」という性質を利用しています。サブシェル内にて、cdコマンドで現在のスクリプト設置ディレクトリから/var/tmpへとカレントディレクトリを移動していますが、サブシェルから抜けたところで元のカレントディレクトリに自動的に戻るため、わざわざcdコマンドで元のスクリプト設置ディレクトリに戻る必要はありません。

こうすれば「cdコマンドで元のディレクトリに戻るのを忘れた」という、よく見られる簡単なバグを防ぐことができます。

注意事項

・サブシェル内で変数の値を変えても、元のシェルには反映されません。サブシェル内で変数の値を変えたつもりになっていたら変わっていなかった、というのはよくあるミスです。

関連項目

012 関数の中でローカル変数を定義して、呼び出し元の変数を破壊しないようにする
023 絶対パスで起動されても相対パスで起動されても、同じ動作をできるようにする
129 中間ファイルを作らずにコマンドの出力をファイルのように扱う

No. 026 ディレクトリ内のファイル数・ディレクトリ数を調べる

利用コマンド
find, wc

キーワード
ファイル数, ディレクトリ数, カウント

いつ使うか
作業ディレクトリなど、たくさんのファイルができるディレクトリの中の、ファイル数・ディレクトリ数をカウントしたいとき

実行例

```
$ ./findcount.sh
対象ディレクトリ: /home/user1/myapp/work
ファイル数: 4
ディレクトリ数: 2
```

スクリプト

```
#!/bin/sh

targetdir="/home/user1/myapp/work"

filecount=$(find "$targetdir" -maxdepth 1 -type f -print | wc -l)
dircount=$(find "$targetdir" -maxdepth 1 -type d -print | wc -l)

dircount=$(expr $dircount - 1)

echo "対象ディレクトリ: $targetdir"
echo "ファイル数: $filecount"
echo "ディレクトリ数: $dircount"
```

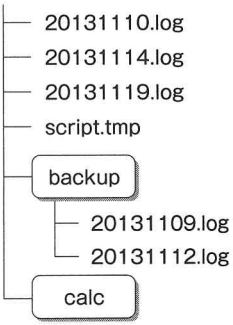
❶
❷

解説

このスクリプトは、/home/user1/myapp/workというディレクトリの中にあるファイルの数とディレクトリの数を、レポートして表示するものです。このレポート内では、指定ディレクトリ直下のファイルのみをカウントし、サブディレクトリの中のファイル数は含めないこととします。

なおここで、対象ディレクトリ/home/user1/myapp/workの中のファイルリストは次のようになっているとします。

対象ディレクトリ以下の構造



すなわちファイルとして、20131110.log、20131114.log、20131119.log、script.tmpの4つがあり、さらに「backup」「calc」という2つのサブディレクトリがあります。また、サブディレクトリ[backup]の中には20131109.logと20131112.logという2つのファイルがあるとします。

さて、特定のディレクトリ配下のファイルリストを作る際には、**findコマンド**がよく使われます。findコマンドの基本的な使い方は、次のようになります。

書式 findコマンドの書式

```
find [対象パス] [式]
```

式としてよく使われるのは、-nameと-typeです。**-name**はファイル名を指定する式で、特定のパターンのファイル名を探索するときに利用します。

一方、ここで用いている**-type**は、ファイルの種別を指定する式です。-typeで指定できる種別は表1のようになります。このうちよく使われるのは、-type d（ディレクトリ）、-type f（通常のファイル）、-type l（シンボリックリンク）でしょう。

findコマンドのファイル種別指定

式の指定	説明
-type b	ブロックスペシャルファイル
-type c	キャラクタスペシャルファイル
-type d	ディレクトリ
-type p	FIFO（名前付きパイプ）
-type f	通常のファイル
-type l	シンボリックリンク
-type s	ソケット

❶では、-type fと-type dでそれぞれファイルリストとディレクトリリストを表示し、その行数を**wcコマンド**の**-lオプション**で取得することで、シェル変数filecountにファイル

数を、シェル変数dircountにディレクトリ数を代入しています。

なお、❶でfindコマンド実行の際、**-maxdepth**というオプションを利用しています。これはサブディレクトリを対象としないための指定です。

findコマンドは何も指定しないと、**サブディレクトリも含めて探索**してしまいます。このサンプル例では、ディレクトリ/home/user1/myapp/work直下のファイル数とディレクトリ数をカウントすることとしており、サブディレクトリは含めない仕様でした。そのため、-maxdepth 1として、1階層(指定ディレクトリの直下)だけを対象にしています。

findコマンドの最後に、ここでは**-print**を用いています。これは見つかったファイルをそのまま表示します。なお、何も指定しないと-printとみなされるため、-printは省略可能です。-printの他にも、-exec(コマンド実行)、-ls(詳細情報付きでファイルリスト表示)などを指定できます。詳しくはman findとしてfindコマンドのマニュアルを読んでみてください。

❷では、カウントしたディレクトリ数から1を引いています。これは、findコマンドの-type dでディレクトリリストを表示すると、次のように対象パスがはじめに表示されるため、実際にあるサブディレクトリの数より1つ多くカウントされてしまうからです。

findコマンドによるディレクトリリスト表示

```
$ find /home/user1/myapp/work -type d -print
/home/user1/myapp/work
/home/user1/myapp/work/backup
/home/user1/myapp/work/calc
```

すなわち、対象パス内のディレクトリを数える際には、-type dの出力結果から1を引かないといけません。そのため❷で、**exprコマンド**を利用してシェル変数dircountから1を引いて、その結果をコマンド置換**$()**で得ています。これでディレクトリ数を数えることができます。

関連項目

031 作業ファイルディレクトリから、1年以上更新のないファイルを削除する
044 2つのディレクトリ内を比較し、どちらか片方だけに存在するファイルを表示する

No. 027 ファイルの中身を消去して、ゼロバイトの空ファイルにする

利用コマンド
:（ヌルコマンド）, uptime, sleep

キーワード
空ファイル, 初期化

いつ使うか
スクリプト起動時に、記録ファイル・一時ファイルを初期化したいとき

実行例

```
$ ./null.sh
```

スクリプト

```
#!/bin/sh

# uptime コマンドの記録ファイルを定義
uptimelog="uptime.log"

# ヌルコマンドで空ファイルに初期化する
: > $uptimelog ―――――――――――――❶

# 10秒おきに6回、uptime コマンドを実行
for i in 1 2 3 4 5 6 ―――――――――――❷
do
  uptime >> $uptimelog ―――――――――❸
  sleep 10
done
```

解説

このスクリプトは、現在のサーバの負荷状態を確認するものです。ロードアベレージ（サーバの負荷）を出力する**uptimeコマンド**の出力結果をログファイルへ10秒おきに6回、つまり1分間のあいだ出力しています。スクリプト実行後、次ページのようにuptime.logにロードアベレージが記録されています。

⬇ロードアベレージのログファイル

```
$ cat uptime.log
 22:26:46 up 6 days, 23:51,  1 user,  load average: 0.10, 0.20, 0.33
 22:26:56 up 6 days, 23:51,  1 user,  load average: 0.10, 0.21, 0.32
 22:27:06 up 6 days, 23:51,  1 user,  load average: 0.11, 0.22, 0.32
 22:27:16 up 6 days, 23:52,  1 user,  load average: 0.13, 0.20, 0.32
 22:27:26 up 6 days, 23:52,  1 user,  load average: 0.15, 0.19, 0.33
 22:27:36 up 6 days, 23:52,  1 user,  load average: 0.18, 0.23, 0.32
```

なお、uptimeコマンドで出力されるロードアベレージ(load average:の右側)は、左から順に過去1分間・5分間・15分間の平均値です。ロードアベレージとは、サーバ上で待ち状態になっているタスクの数を表した数値です。この値が大きければ、待ち状態となっているタスクが多い、すなわちCPUやディスクなどのリソースが不足しておりサーバの負荷が高い状態であると判断できます。そこでサーバの負荷計測などのために、サンプル例のようなスクリプトで、負荷データを取得しておくケースが考えられるでしょう。

この例のように、スクリプトの実行中に、状態の記録や保存をするために作業ファイルを利用するプログラムは多くあります。その際に気を付けなければいけないのは、**前回起動時のファイルが残っていたり、異常終了時に作業ファイルの残骸ができてしまう**かもしれないということです。

そのようなトラブルを避けるためには、スクリプトのはじめに空っぽの作業ファイルをまず作成してしまい、環境を初期化するのが1つの手です。ファイルを空にするには、このスクリプト例のように何も出力しない**:(ヌルコマンド)をリダイレクト**する方法が簡単でよく使われます。

❶が、:(ヌルコマンド)を利用した空ファイル作成を行っている部分です。ここでは:の出力結果をリダイレクトしています。:は何も出力しないので、ファイルがなければ空っぽのファイルが作成されますし、既存ファイルがあればその中身をクリアして空っぽにしてくれます。そのため、すでにログファイル(ここではuptime.log)が存在するかどうかは気にする必要がありません。

❷でuptimeコマンドを繰り返すためのfor文を書いています。ここでは6回繰り返すことにしたので、リストとして「1 2 3 4 5 6」を与えています。これは単なるループのカウンタなので、6個の引数があることだけに意味があり、値自体には特に意味はありません。「a b c d e f」としてもかまわないわけですが、慣習的にループには数値を用います。

❸でログファイルにuptimeコマンドの出力結果を記録しています。ここではループごとに内容が上書きされないよう、追記のリダイレクト>>を用います。この後にsleepコマンドで10秒間待って、再びuptimeコマンドを打つことを6回繰り返しています。

なお、新規ファイル作成ならばtouchコマンドを使えばよいと思われるかもしれません。しかしtouchコマンドは既存ファイルがある場合は更新日を変更するだけで、空っぽのファイルを作成することができず初期化には使えません。

既存ファイルが存在する場合は空ファイルとならない

```
$ cat uptime.log
 22:26:46 up 6 days, 23:51,  1 user,  load average: 0.10, 0.20, 0.33
$ touch uptime.log
$ cat uptime.log
 22:26:46 up 6 days, 23:51,  1 user,  load average: 0.10, 0.20, 0.33
```

空ファイルを作るその他の方法

空っぽのファイルを作るには、:(ヌルコマンド)を使う以外にもいくつかの方法があるため、ここでいくつか紹介します。

次のように**/dev/null**をコピーする手法が一般的です。

```
cp /dev/null $uptimelog
```

また/dev/nullをcatして出力リダイレクトする手法もよく使われます。

```
cat /dev/null > $uptimelog
```

/dev/nullはUNIXで用いられるスペシャルファイルで、中身を読み出すとEOF(エンドオブファイル)が出力されます。そのため、この中身をファイルに書き出すことで空ファイルを得ることができます。

またtrueコマンドを使っても同じ結果が得られます。

```
# uptime.log を空ファイルに初期化
true > uptime.log
```

trueコマンドは:(ヌルコマンド)と同じく、何も出力せずに終了ステータス0を返します。trueコマンドは外部コマンドであるため、サンプル例のようなケースでは、内部コマンドである:(ヌルコマンド)を用いたほうがよいでしょう。

関連項目

089 異常終了してもゴミが残らないよう、終了前に作業ファイルを消去して後始末を行う

118 CPU使用率の監視を行う

No. 028 新規ファイルを作らずに、すでにあるファイルのみファイル更新日を変更する

利用コマンド

touch

キーワード

タイムスタンプ, 新規ファイル, 更新日

いつ使うか

touchコマンドでタイムスタンプを変更する初期化スクリプトなどで、存在しないファイルは新規作成したくないとき

実行例

```
$ ./touch.sh
```

スクリプト

```
#!/bin/sh

# 〔YYYYMMDDhhmm.SS〕として、〔年月日時分.秒〕を指定
timestamp="201311190123.45"

# ファイルのタイムスタンプ更新。
# -cオプション付きのため、ロックファイルは新規作成はしない
touch -t $timestamp app1.log ―――――❶
touch -c -t $timestamp lock.tmp ―――――❷
```

解説

このスクリプトは、app1.logとlock.tmpという2つのファイルのタイムスタンプを更新するものです。想定シーンとしては、ログファイル操作を行う別のプログラムのために、テストデータを作りたいケースを考えることとします。この別のプログラムは、ログファイル(app1.log)のタイムスタンプを判別して、何らかの処理を行うものです。

1つテストを終えるたびにシェルスクリプトを実行して、テストデータを初期化する、というのはよく使われる手法です。このサンプル例は、そんなテストの一環で書かれたシェルスクリプトです。

このスクリプトでは、タイムスタンプの操作に**touchコマンド**を利用しています。ここでまず、touchコマンドの解説の前に、UNIXにおけるファイルのタイムスタンプの仕組みについて少し見ておきましょう。

一般にUNIXでは、ファイルのタイムスタンプには次の3つの種類があります。ファイルの内容を修正すると、mtimeが更新されます。なおここでctimeは、create time(ファイル作成日時)ではなくchange time(状態変更日時)であることに注意してください。

UNIXにおけるファイルのタイムスタンプ

種類	説明
atime	最終アクセス時刻 (access time)
mtime	最終修正時刻 (modify time)
ctime	最終状態変更時刻 (change time)

これらタイムスタンプは、次のように**statコマンド**を使って詳しく見ることができます。

statコマンドでタイムスタンプを確認する (Linuxの場合)

```
$ stat touch.sh
  File: `touch.sh'
  Size: 261             Blocks: 8          IO Block: 4096   regular file
Device: fc03h/64515d    Inode: 3276911     Links: 1
Access: (0775/-rwxrwxr-x)  Uid: ( 1003/   user1)   Gid: ( 1003/   user1)
Access: 2013-04-10 11:15:13.000000000 +0900
Modify: 2011-07-17 18:52:26.000000000 +0900
Change: 2013-11-11 22:58:16.461418708 +0900
```

なおstatコマンドを見やすくするために、FreeBSDおよびMacの場合は**-xオプション**を付けてください。

statコマンドでタイムスタンプを確認する (Macの場合)

```
$ stat -x touch.sh
  File: "touch.sh"
  Size: 309           FileType: Regular File
  Mode: (0777/-rwxrwxrwx) Uid: (  501/TechnicalBook)  Gid: ( 20/ staff)
Device: 1,2   Inode: 11630161    Links: 1
Access: Thu May 22 18:21:10 2014
Modify: Fri Dec 13 15:03:04 2013
Change: Thu May 22 18:21:10 2014
```

statコマンド実行例の最後の3行、Access/Modify/Changeがそれぞれatime/mtime/ctimeに該当します。今回のスクリプトが対象とする、ファイルのタイムスタンプを見て動作するプログラムでは、最終修正時刻 (mtime) を見て判断するものが多いでしょう。

❶で利用しているtouchコマンドは、**-tオプション**で時刻を指定するとファイルのatimeとmtimeを更新します (加えて-aオプションおよび-mオプションを利用すると、atimeもしくはmtimeのどちらか片方だけ更新することも可能です)。なお-tオプションは、指定する時刻 [年月日時分.秒] を [YYYYMMDDhhmm.SS] として指定します。つまり❶は、「2013年11月19日1時23分45秒」を指定してファイルのタイムスタンプatimeとmtimeを更新しています。

続いて❷で、ロックファイルのタイムスタンプも変更しています。ここでロックファイルを操作する際に、**-cオプション**を付けています。これはファイルがない場合には新しくファイルを作らないようにするための処理です。

touchコマンドを利用すると空ファイルが作成できる、というのはよく知られた動きです。しかしこのようにテストファイルの初期化を行う場合は、「ファイルがあればタイムスタンプを変更したいけれど、ファイルがない場合は何もしない(ファイルを作らない)」としたいことがよくあります。

この場合、次のようにいちいちファイルの存在をif文で確認するのは読みにくいですし、後で動作を変えたいときに修正するのも大変です。

```
if [ -e lock.tmp ]; then
  touch -t $timestamp lock.tmp
fi
```

それよりも、サンプル例のようにtouchコマンドの-cオプションを利用すれば、存在しないファイルは作らないようにすることができるため、簡潔にわかりやすく書くことができます。

注意事項

- touchコマンドを-tオプションを付けずに実行すると、ファイルのタイムスタンプはコマンドの実行日時で更新されます。ファイルの内容は変更しないけれども、タイムスタンプだけ変更したい、という処理でよく使われます。

```
#ファイルのタイムスタンプを「いま」にする
touch app1.log
```

- タイムスタンプを利用してファイルを検索するには、findコマンドの**-mtimeオプション**を用いる例がよく使われます。

関連項目

030 あるディレクトリ内のn日前からm日前までに更新されたファイル一覧を取得する
031 作業ファイルディレクトリから、1年以上更新のないファイルを削除する

No. 029 複数HTMLファイルからtitleタグ部分のみを抜き出して、それぞれ別ファイルへ出力する

利用コマンド
basename, sed

キーワード
for文, ファイルリスト, HTMLのタグ, 別ファイル

いつ使うか
多数のHTMLファイルから特定の要素を抽出し、HTMLファイルごとに別々のファイルへ出力したいとき

実行例

```
$ ls output/
$ ./htmltitle.sh
$ ls output/
about.txt  index.txt  menu.txt
```

スクリプト

```
#!/bin/sh

# カレントディレクトリの .html ファイルを対象
for htmlfile in *.html ―――――――――――――――――❶
do
  # ファイル名から、拡張子を含まない文字列を取得する
  fname=$(basename $htmlfile .html) ―――――――――❷

  # <title>タグの中身を後方参照¥1として抽出し、ファイル出力する
  sed -n "s/^.*<title>¥(.*¥)<¥/title>.*$/¥1/p" $htmlfile > output/${fname}.
txt ―――――――――――――――――――――――――――❸
done
```

解説

このスクリプトは、カレントディレクトリのHTMLファイル(拡張子が.htmlのファイル)から<title>タグを抽出し、そのtitle要素をそれぞれ別のファイルとしてoutputというディレクトリへ出力するものです。例えばindex.htmlのtitle要素はindex.txtへ、menu.htmlのtitle要素はmenu.txtへと出力します。

❶で、HTMLファイルを順に処理するためにfor文を使っています。シェルスクリプトでディレクトリ内のファイルを順に処理したい場合は、この❶のように**パス名展開**する方法が簡単でよく使われます。for文のinの後ろに、*.htmlというパターンを与えれば、実

行時には次のようにパス名展開され、シェル変数htmlfileを用いて各ファイルを順に処理することができます。

```
for htmlfile in index.html about.html menu.html
do
  ...
```

このようにパス名展開の結果をfor文のリストとして与える手法はいろいろと応用が効くため、よく使われる手法です。

続いて、各ファイルを.txtとして出力するため、元のファイル名から拡張子を変更した出力用のファイル名を組み立てる必要があります。その準備として、❷でまず対象htmlファイルのファイル名から、拡張子を取り除いた文字列を取得しています。これには**basenameコマンド**(→P.64)を利用しています。

このコマンド出力により得られた文字列の後ろに、任意の拡張子を付けてファイル名を組み立てれば、拡張子を変更して出力できるわけです。このスクリプトでは、.txtを付けています。

❸が、<title>タグの中身をパターンマッチで抽出してファイルに出力する処理です。パターンマッチした部分を取り出すにはさまざまな手法がありますが、ここでは**sedコマンド**の**-nオプション**(パターンスペースを出力しない)と**pフラグ**(置換が発生した場合のみ出力する)を組み合わせて、後方参照¥1により<title>タグの中身を取り出しています。このsedコマンドの使い方の詳細は、P.46の例を参考にしてください。

注意事項

- このスクリプトでは、htmlファイルが存在するかどうかのチェックは省略しているため、カレントディレクトリにhtmlファイルが1つもない場合はエラーになります。

関連項目

013 読み込んだHTMLファイルから特定の属性値を取得する
018 HTMLファイルから、タグの中に書かれたコマンドを抜き出してそのまま実行する
041 拡張子に.htmと.htmlが混じったHTMLファイル群の拡張子を一括してtxtに変更する

No. 030 あるディレクトリ内の、n日前からm日前までに更新されたファイル一覧を取得する

利用コマンド
find

キーワード
前日, 更新日, 日付, タイムスタンプ, mtime

いつ使うか
特定の期間に作成・更新されたファイルをリストアップしたいとき

実行例

```
$ ./find-mtime.sh ―――― 現在の時刻が2013年11月26日 20:00とすると…
/var/log/myapp/201311222346.log
/var/log/myapp/201311230446.log
/var/log/myapp/201311230946.log
/var/log/myapp/201311240046.log
/var/log/myapp/201311240546.log
/var/log/myapp/201311242046.log
```

スクリプト

```
#!/bin/sh

logdir="/var/log/myapp"

# 4日前から2日前までに更新されたファイル一覧を表示する
find $logdir -name "*.log" -mtime -4 -mtime +1 -print
```

解説

このスクリプトは、シェル変数logdirで指定されたディレクトリの中から、4日前から2日前までのあいだに更新のあったログファイル(拡張子が.logのファイル)一覧を表示するものです。ここで、ディレクトリ/var/log/myappには、ファイル変更日時をファイル名に持つログファイルがたくさんあるものと仮定しています。ファイル検索の際、拡張子の指定には、findコマンドの-nameを利用して"*.log"としています。findコマンドの基本的な使い方は、P.68を参照してください。

このサンプル例では、**findコマンド**でファイルの変更日時を対象とする**-mtime**を利用していることがポイントです。つまりファイル名の「日」ではなくファイルが持つタイムスタンプを使用してファイルを検索しています。

まずはじめに予備知識として、findコマンドの-mtimeでの日数の数え方を説明します。

findコマンドの-mtimeは日数nを指定しますが、ここで指定する「n日前」とは、「n×24

時間前」を意味します。例えば現在の時刻が11月25日の10時30分ならば、「1日前」とは11月24日の10時30分までを指し、それより前（例えば25時間前）の11月24日9時30分は前々日ということになります。n日と指定したとき、これは**カレンダー上の日付でのn日ではない**ということにまず注意してください。

さて、findコマンドの-mtimeには日数を指定しますが、この際にプラスを付けるかマイナスを付けるかで意味が違ってきます。マイナスを付けた-nで「n日前より新しい」、符号なしのnで「n+1日前からn日前」、プラスを付けた+nで「n日よりも過去（「n日と数時間前」のファイルは端数時間を切り捨ててn日扱いになるので、実質的にはn+1日よりも前）」という意味になります。繰り返しになりますが、ここでn日前とはn×24時間前であることに注意してください。

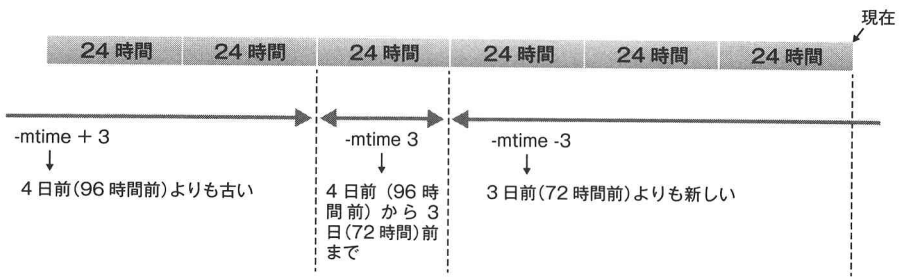
例えばn=3の場合は、次の表のようになります。

◆日数の指定例

日数の指定例	説明
find -mtime -3	3日（72時間）前よりも新しい
find -mtime 3	4日（96時間）前から3日（72時間）前まで
find -mtime +3	4日（96時間）前よりも過去

これは文章だけですと非常にややこしいので、次の図も参考にしてください。

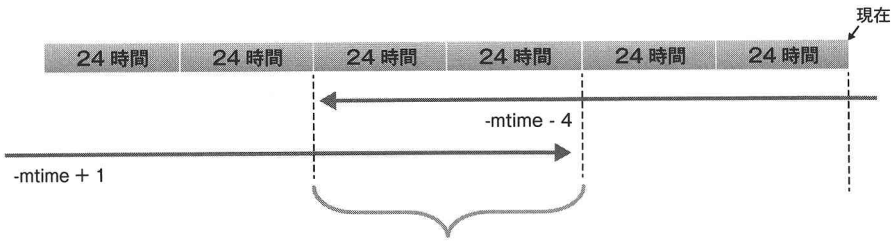
◆指定と実際の期間



このように-mtimeに指定した日数と、プラスとマイナスの付け方で期間が変わってきますので注意してください。

サンプル例では、この-mtimeのプラス指定とマイナス指定を組み合わせることで、「4日前から2日前までのあいだに変更のあったファイル」を抽出しているのです。

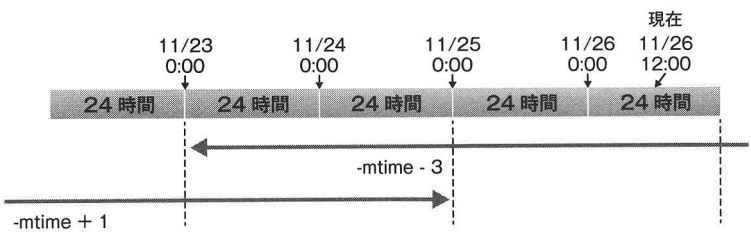
複数の-mtimeで期間を指定する



注意事項

- FreeBSDでは、-mtime +nのときに端数の時間を切り上げで数えるため、Linuxの場合と若干動作が違います。具体的には、-mtime +3で「3日前よりも過去」を示します。ここでの説明から1日ずれることに注意してください。
- -mtime -nのマイナス指定は、よく「n日以内」という書き方をされ、実用上はそれでほとんど問題ありませんが、正確には「n日前よりも新しい」ファイルです。つまり、もし現在よりも未来の日付のファイルがあった場合、対象となります。未来の日付のファイルはtouchコマンドの-tオプションで作成できます(→P.74)。
- MacやFreeBSDのfindコマンドでは使えませんが、Linuxのfindコマンドには、**-daystart**というオプションがあります。これを用いると、日数の数え方を現在の時刻と関係なく「現在日の0時0分から」と扱ってくれます。

-daystartオプションを指定したとき



-daystartオプションを付けたほうが、カレンダーの日付と一致させて「昨日」「n日前」が指定できてわかりやすくなります。ただし、MacやFreeBSDでは使えないため、移植性を損なうことには注意が必要です。

関連項目

028　新規ファイルを作らずに、すでにあるファイルのみファイル更新日を変更する
031　作業ファイルディレクトリから、1年以上更新のないファイルを削除する

No. 031 作業ファイルディレクトリから、1年以上更新のないファイルを削除する

利用コマンド
find, xargs

キーワード
更新日, 日付, ファイル削除, 自動削除

いつ使うか
長いあいだに変更のなかったファイルや古いログファイルを削除したいとき

実行例

```
$ ./find-del.sh ―――― 現在の日付が2013年11月26日とすると
/var/log/myapp/201211250147.log
/var/log/myapp/201211200147.log
/var/log/myapp/201211150147.log
```

スクリプト

```
#!/bin/sh

logdir="/var/log/myapp"

# 最終変更日時が1年以上前の古いファイルを削除する
find $logdir -name "*.log" -mtime +364 -print | xargs rm -fv ――❶
```

解説

このスクリプトは、シェル変数logdirで指定されたディレクトリから、1年(365日)以上変更のないログファイルをfindコマンドで見つけて削除するものです。ここで、ディレクトリ/var/log/myappには、ファイル変更日時をファイル名に持つログファイルがたくさんあるものと仮定しています。このサンプルでは、ファイルリストをxargsコマンドで処理しているのがポイントです。

また、この例では、1年以上変更のないファイルを、**findコマンド**の-mtimeを利用して取得しています。findコマンド、および-mtimeオプションの詳細は、P.68とP.79を参照してください。

このスクリプトで利用している**xargsコマンド**は、ファイルリストを引数として受け取り、任意のコマンドを実行するためによく使われます。主な使い方としては、findコマンドで特定の条件にマッチするファイルリストを出力し、それをパイプでxargsコマンドが受け取って処理するという使い方が一般的です。

書式 findとxargsの連携プレー

```
find <対象パス> <式> | xargs <実行したいコマンド>
```

❶では、まずシェル変数logdirで指定されるディレクトリから、-nameで拡張子が.logのファイルを取得しています。この際、**-mtime**を併用することで、365日以上前が変更日となっているファイルを選択しています。こうして条件にマッチするファイルリストを出力してxargsコマンドに渡し、ファイルを削除するrmコマンドを実行しています。**rmコマンド**には、該当ファイルが1つもないときにもエラーとならないように**-fオプション**を付けて、また同時に**-vオプション**を付けて削除したファイル名を表示するようにしています。

さて、このサンプルのように古いファイルを削除するスクリプトは、長いあいだ稼働させるWebアプリケーションなどの補助バッチとしてよく使われます。

一時ファイルや動作ログファイルをとりあえずどこかに出力しておくという作りのアプリケーションは開発途中などでよく見られます。何年も稼働するシステムでは、この一時ファイルがそのまま放置されることがあり、扱いに注意が必要です。具体的には、以下のようなケースです。

1) 初期バージョンでは、開発者がデバッグなどのためにとりあえず一時ファイルを出力しておく実装にしてそのままリリースされる
2) 元の開発者はいなくなり、運用担当者がサーバごと担当を引き継ぐ
3) 数年経った頃、削除されない一時ファイルがどんどんディスク領域を食っていき、いつの間にか使用率100%になりシステムがダウンする

このようなことは、(残念ながら)あちこちの現場でよく見られる光景です。このような事態は、システムリリース時にはじめから、古いファイルは削除するような考慮をしておけば防ぐことができるはずです。

サイズの大きな一時ファイル・ログファイルを作成するアプリケーションを作る際は、将来ディスクがあふれてしまって大きな事故となるのを防ぐために、このように古いファイルを自動削除するバッチを作っておくとよいでしょう。

注意事項

・いきなりファイルを削除するスクリプトを書くのは危険なので、まずは意図どおりに動いているかを確認するようにしましょう。具体的には次のように、xargsで実行するコマンドをlsにして、試してみるとよいでしょう。

```
find $logdir -name "*.log" -mtime +364 -print | xargs ls
```

こうすれば、実際にxargsでコマンドが実行される対象のファイルリストのみを表示でき

ます。意図しないファイルが対象になっていないかを確認してから、実行するコマンドをlsからrmに修正します。

- テストの際に更新日付が古いファイルを作るためには、touchコマンドの-tオプションが利用できます。P.74を参照してください。
- ファイル名に空白文字（スペース）を含む場合は、このサンプル例ではエラーとなります。そのような場合は、文字列の区切りに空白ではなくヌル文字が使われているとみなす、xargsコマンドの-0オプションを利用します。この際、findコマンドのほうも、区切りをヌル文字として出すように-print0オプションを使います。

```
find $logdir -name "*.log" -mtime +364 -print0 | xargs -0 rm -fv
```

こうすれば、空白文字を含むファイル名も正しく扱うことができます。

関連項目

028 新規ファイルを作らずに、すでにあるファイルのみファイル更新日を変更する
030 あるディレクトリ内の、n日前からm日前までに更新されたファイル一覧を取得する
032 大量のログファイルがあるディレクトリ内のファイルに一括したコマンドを実施する

No. 032 大量のログファイルがあるディレクトリ内のファイルに一括したコマンドを実施する

利用コマンド
find, xargs, grep

キーワード
引数, コマンドライン引数, 大量ファイル

いつ使うか
大量のファイルがあり、単純に*でファイルを指定するとエラーになる場合に、grepコマンドなどを実行したいとき

実行例

```
$ ./xargs-grep.sh
/var/log/myapp/49294.log:2012-12-24 00:04:59 [ERROR] File Not Found.
/var/log/myapp/23100.log:2013-06-10 03:54:21 [ERROR] I/O Error.
/var/log/myapp/14322.log:2013-10-12 13:21:03 [ERROR] File Not Found.
/var/log/myapp/21322.log:2013-10-12 13:21:04 [ERROR] File Not Found.
```

スクリプト

```
#!/bin/sh

logdir="/var/log/myapp"

# 拡張子.logのファイルから、"ERROR"という文字列を検索
find $logdir -name "*.log" -print | xargs grep "ERROR" /dev/null —❶
```

解説

このスクリプトは、大量のファイルが置かれているディレクトリ/var/log/myappに対して、"ERROR"という文字列を含むファイルをgrepコマンドで検索するものです。ここでディレクトリ/var/log/myappは、単純にgrepコマンドを実行するとエラーになるほどの、大量のログファイルが置かれていることを仮定しています。

大量のファイルがあるとエラーが生じることがある

```
$ grep 'ERROR' *.log
-bash: /bin/grep: Argument list too long
```

ここで、"Argument list too long"（OSによって多少メッセージは違います）と表示されてエラーになる理由は、*（アスタリスク）がシェルによって展開されたときにとても長い

文字列となるために、コマンドライン引数が、システムが扱える上限を超える長さとなってしまうことにあります。

UNIXでは、コマンドライン引数の上限値は**ARG_MAX**定数で決め打ちされています。そのため大量のファイルがある場合に*でファイルリストを与えると、ARG_MAX以上の文字列長となりエラーになります。このARG_MAXの値は、例えばLinuxならば次のように**getconfコマンド**で確認できます。

コマンドライン引数の上限値を確認

```
$ getconf ARG_MAX
2621440
```

興味のある方は、man execveとしてexecve(2)のマニュアルを読んでみてください。execveはプログラムを実行するシステムコールで、例えばLinuxならばARG_MAXの値が<limits.h>で定義されていることなどが解説されています。

この制限を回避するには、サンプル例のように**findコマンド**でまずファイルリストを出力し、それを**xargsコマンド**で受け取って**grep**を実行するというのがよく使われる手法です。findコマンドの基本的な使い方は、P.68を参照してください。

xargsコマンドは、ARG_MAXの値を超えないように引数を適宜区切って、指定されたコマンドを実行します。そのためコマンドライン引数がどれだけ長くても、ARG_MAXの制限に引っかからないように正しく処理することができるのです。

なお、このスクリプトではちょっとした技として、対象のファイルに/dev/nullを加えています。これは、grepコマンドの出力に必ずファイル名を含むようにするための処理です。grepコマンドでは複数のファイルを対象とした場合、次のように先頭にファイル名を付けてマッチした行を出力します。

複数ファイルを対象としたgrepコマンド

```
$ grep "ERROR" *
<ファイル名>:<マッチした行>
<ファイル名>:<マッチした行>
<ファイル名>:<マッチした行>
...
```

このサンプル例では、/var/log/myappに大量のファイルがあると仮定していました。しかし、もし対象ファイルが1つしかなかった場合には、grepコマンドは結果にファイル名を出力しません。

対象ファイルに*と指定したが、実は1つのファイルしかない場合

```
$ grep "ERROR" *
<マッチした行>
```

これでは対象のファイル数によって結果出力が異なってしまうため、後処理する際にも不便です。そこでこのサンプル例では、対象のファイルに決め打ちで/dev/nullも加えておくことで、grepコマンドが常に複数ファイルを対象とするようにして、結果にファイル名が表示されるようにしているのです。

/dev/nullはどんな文字列も含まれることはないので、grepに引っかかることもなく、検索結果には影響を与えません。

注意事項

- この例では、ファイルに空白文字（スペース）を含む場合はエラーとなってしまいます。空白文字を扱うには、findコマンドで-print0オプションを利用する必要があります。P.84の注意事項を参照してください。

関連項目

030 あるディレクトリ内のｎ日前からｍ日前までに更新されたファイル一覧を取得する
031 作業ファイルディレクトリから、1年以上更新のないファイルを削除する

No. 033 ファイルをバックアップする際にファイル名に日時を入れる

利用コマンド
date, cp

キーワード
バックアップ, 現在時刻, 日付

いつ使うか
あるファイルのバックアップを取得する際、現在の日付を入れて簡単にコピーをとりたいとき

実行例

```
$ ls ―――――――――――――――――――――――――――― ファイルの確認
datename.sh      myapp.conf
$ ./datename.sh ―――――――――――――――――――――― 実行
myapp.conf -> myapp.conf.20131202
$ ./datename.sh ―――――――――――――――――――――― 実行
myapp.conf -> myapp.conf.201312022255.20
$ ls ―――――――――――――――――――――――――――― ファイルの確認
datename.sh  myapp.conf  myapp.conf.20131202  myapp.conf.201312022255.20
```

スクリプト

```
#!/bin/sh

config="myapp.conf"

bak_filename="${config}.$(date '+%Y%m%d')" ―――――――― ❶

# すでにmyapp.conf.20131202 などがあった場合は、秒まで入れて
# バックアップファイルを作成する
if [ -e "$bak_filename" ]; then
  bak_filename="${config}.$(date '+%Y%m%d%H%M.%S')" ―― ❷
fi

cp -v "$config" "$bak_filename" ―――――――――――――― ❸
```

解説

このスクリプトは、カレントディレクトリにあるmyapp.confというコンフィグファイルのバックアップを取得するものです。実行すると、現在の日付を利用してファイル名を組み立て、「myapp.conf.20131202」と、「元のファイル名＋日付」としてコピーを作ります。

なお、同一日に再度このスクリプトが叩かれた場合は、バックアップファイルを上書き

しないよう、時分秒までも含めて「myapp.conf.201312022255.22」というファイル名でバックアップを作ります。

このスクリプトでは現在日付を取得するために、**dateコマンド**を利用しています。dateコマンドは、引数が+で始まる場合には現在の時刻(日付)を指定した形式で表示します。この時刻表示形式は、引数としてフィールド名を与えることで制御でき、年月日や時分秒を自由に組み合わせて文字列を作ることができます。

dateコマンドでよく使われるフィールド名を、次の表に示します。これらのフィールドはライブラリ関数strftimeで定義されているため、詳しくはman strftimeを参照してください。

dateコマンドのフィールド

フィールド名	説明
%Y	年(1970～)
%y	年の下2桁(00～99)
%m	月(01～12)
%d	日(01～31)
%H	時(00～23)
%M	分(00～59)
%S	秒(00～59)

❶では、**$()**という**コマンド置換**を利用してファイル名を組み立てています。この際、dateコマンドの表示形式として%Y%m%d、すなわちYYYYMMDD(年月日)を利用しています。いまが2013年12月2日ならば、これは"20131202"という文字列になります。これにより、現在の日付を後ろに付けてバックアップファイル名を組み立てているわけです。

なおこのままでは、同一日に2回以上このスクリプトが実行された場合は前のファイルを上書きしてしまうため、❷で上書きチェックの処理を加えています。もし現在日付が末尾に付いたファイルがすでにある場合は、上書きしないように時分秒まで含んだファイル名でコピーしています。時分秒を指定するには、フィールドとして%H%M%Sを利用します。ここでは見やすくするために、秒の前にピリオドを打っています。これにより、2013年12月2日22時55分22秒にスクリプトが実行されると、"myapp.conf.201312022255.22"というファイル名を組み立てます。

❸でバックアップのためにファイルをコピーしていますが、ここでは**cpコマンド**に**-vオプション**(verboseオプション)を付けています。これにより、どのファイルを何というファイル名でコピーしたかを表示することができます。ファイル操作系のシェルスクリプトでは、画面で見て(あるいは後でログファイルを見て)確認できるように、このように-vオプションを付けておくと便利でしょう。

注意事項

・このスクリプトは秒でファイル名を組み立てるため、1秒以内に何度も実行されるとファイ

ルを上書きしてしまいます。慎重を期するなら、すでにファイルがある場合は、後ろに1、2、3……と数値を付けていく方式も考えられます。しかし、この方式は同一日内のどの時刻で変更されたかがひと目でわかりにくいことや、ここでの「設定ファイルのバックアップ」という用途ではそうそう1秒以内に更新されることもないため、このサンプル例ではすでにファイルがある場合は、さらに時分秒を付ける形式を採用しました。

- 日付でバックアップファイルを作るのは簡単な方法でよく使われますが、ある意味、場当たり的な方法でもあります。本来ならば設定ファイルは、SubversionやGitなど、バージョン管理システムで世代管理するのが望ましい手法です。
  Subversionはsvnコマンド、Gitはgitコマンドで利用することができます。例えば以下はSubversionでバージョン管理しているファイルの履歴を表示したものです。いつ誰がどう編集したのか一目でわかりますし、過去のバージョンに簡単に戻すこともできます。SubversionやGitに興味を持たれた方は、それぞれの専門書を参照してください。

Subversionでファイル履歴を表示する

```
$ svn log myapp.conf
------------------------------------------------------------------------
r194 | ozuma | 2014-01-19 18:49:21 +0900 (日, 19  1 2014) | 2 lines

server1が故障したため、接続先IPアドレスをserver3のものに変更
------------------------------------------------------------------------
r182 | mollifier | 2014-01-07 10:17:14 +0900 (火, 07  1 2014) | 2 lines

ログ出力先ディレクトリを、/var/log/myappから/disk/log/mayppに変更
------------------------------------------------------------------------
```

関連項目

028 新規ファイルを作らずに、すでにあるファイルのみファイル更新日を変更する

No. 034 ファイル群を別ディレクトリに同期するバックアップ処理を行う

利用コマンド
rsync

キーワード
バックアップ, 同期, 差分, リモートバックアップ

いつ使うか
毎日新しいログファイルができるなど、ファイルが増えていくディレクトリのバックアップを効率的に行いたいとき

実行例

```
$ ls /home/user1/myapp/log ──────────── ログファイルを確認
20131201.log  20131202.log  20131203.log
$ ./rsync.sh
sending incremental file list
log/
log/20131203.log

sent 428 bytes  received 35 bytes  926.00 bytes/sec
total size is 652  speedup is 1.41
```

スクリプト

```
#!/bin/sh
log_dir="/home/user1/myapp/log"
backup_dir="/backup/myapp"

# /home/user1/myapp/log の中のログファイルを、
# /backup/myapp/log ディレクトリにコピーする
rsync -av "$log_dir" "$backup_dir"
```

解説

このスクリプトは、3つのログファイル(20131201.log, 20131202.log, 20131203.log)が格納されているディレクトリ"/home/user1/myapp/log"の中のファイル群を、ディレクトリ"/backup/myapp/log"にバックアップするものです。ここではrsyncコマンドを利用し、大量のファイルがあっても差分ファイルのみ更新を行って効率的にバックアップできることがポイントです。

実行例では、3つのログファイルがあるはずなのに20131203.logしかコピーされていません。これは前回実行時に20131201.log, 20131202.logはすでにコピーされており、今回新規にできた20131203.logのみコピーされた、ということを仮定しています。

このように差分コピーをしてくれる動作は、毎日ログファイルができていくシステムな

どのように、追加で新規ファイルが作られるディレクトリのバックアップに役立つでしょう。

このサンプルで利用している**rsyncコマンド**とは、その名のとおりファイルをsync（同期）するために使われるコマンドです。以下のような特徴を持ち、サーバ管理用途に広く使われています。

・コピー時には、コピー元とコピー先の差分を元に、変更のあったファイルのみコピーするため効率的
・ファイルのタイムスタンプ・パーミッション・所有者情報などのファイル属性をそのままコピーすることができる
・sshを利用してリモートサーバからのコピーも行える

rsyncコマンドの使い方は、次のようになります。

書式 rsyncコマンドの書式

```
rsync [オプション] <コピー元> <コピー先>
```

サンプル例で利用しているオプションは、**-a**（アーカイブモード）と**-v**（verboseモード）です。アーカイブモードとはよく使われるオプションをまとめたもので、先にあげた特徴（ファイルのタイムスタンプや、パーミッション・所有者情報をそのままコピーする）を利用できます。verboseモードは実際にコピーを行ったファイルリストや転送量を表示するオプションで、コマンド実行の結果が目で見てすぐわかりますから、できるだけ付けておいたほうがよいでしょう。

もう1つ覚えておいてほしいオプションが、**-n**（dry-runモード）です。次のように-nを付けると、実際のファイルコピーは行わずに、処理される対象のファイルリストのみが出力されます。これで、実際のファイルコピー対象を確認できます。

```
rsync -avn /home/user1/myapp/log /backup/myapp
```

つまり、シェルスクリプトを書いている段階では上記のようにrsyncする部分は-nオプションで書いておきます。これならば実際のファイルのコピーは行われませんから、テストする際にも気軽に実行することができます。目的どおりのファイルが対象となっていることを確認したら、最後に-avnと指定しているオプションを-avに修正して、-nオプションをやめればよいわけです。

なおrsyncコマンドでは、コピー元ディレクトリの指定時、最後にスラッシュを付けるか付けないかで大きく意味が変わることに注意してください。例えば次のように、コピー元のディレクトリの最後にスラッシュを付けて書いたとします。

```
rsync -avn /home/user1/myapp/log/ /backup/myapp
```

rsyncコマンドでは、**コピー元の最後がスラッシュで終わる場合には「ディレクトリ自身はコピーせず、そのディレクトリの中のファイル・サブディレクトリすべて」**を意味します。そのため、上記はlogディレクトリをコピーするのではなく、logディレクトリの中身(20131201.log, 20131202.log, 20131203.log)のファイルをコピーしようとします。もし上記のようにコピー元のディレクトリの最後にスラッシュを付けて「ディレクトリ自体はコピーしない」とする場合は、コピー先にもディレクトリを指定しないといけません。

このように、rsyncコマンドではディレクトリ指定に若干の注意が必要です。混乱を招かないように、コピー元をスラッシュで終える書き方はせず、本サンプルのように「コピー元はディレクトリ指定することとし、最後にスラッシュは付けない」など事前に何かしらのポリシーを決めておいたほうがよいでしょう。

注意事項

- rsyncコマンドでは、リモートサーバへ(もしくはリモートサーバから)バックアップを取得することができます。この際には、次のようにファイルパスの前に「**ユーザ名@ホスト名:**」を付けます。

```
rsync -av /home/user1/myapp/log user1@server1:/backup/myapp
```

デフォルトでは通信プロトコルはsshが利用されます。また、明示的にsshプロトコルであると指定したい場合は、-e sshオプションを利用します。

```
rsync -av -e ssh /home/user1/myapp/log user1@server1:/backup/myapp
```

- rsyncコマンドでは差分更新を行いますが、コピー元で削除されたファイルをコピー先で消すことはしません。このようなときに完全にディレクトリを同期させたい(すなわち、コピー元でファイルが削除されていたらコピー先からも消したい)ときは、次のように**--deleteオプション**を利用します。

```
rsync -av --delete /home/user1/myapp/log /backup/myapp
```

関連項目

035 ローカルディスクに実ファイルを作らず、直接リモートホストにアーカイブする

No. 035 ローカルディスクに実ファイルを作らず、直接リモートホストにアーカイブする

利用コマンド
tar, ssh, cat

キーワード
tarアーカイブ, リモートホスト, 中間ファイル

いつ使うか
tarアーカイブを作成してリモートホストにコピーする際、中間ファイルを作らずに直接コピーしたいとき

実行例

```
$ ./tar-ssh.sh
myapp/log
myapp/log/20131201.log
myapp/log/20131202.log
```

スクリプト

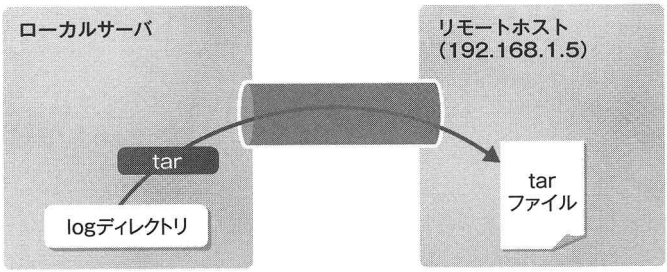
```
#!/bin/sh

username="user1"
server="192.168.1.5"

tar cvf - myapp/log | ssh ${username}@${server} "cat > /backup/myapplog.
tar" ―――――――――――――――――――❶
```

解説

このスクリプトは、ログファイルが格納されたmyapp/logというディレクトリをtarアーカイブし、そのtarアーカイブファイルを192.168.1.5という別のサーバの/backupディレクトリにコピーするものです。この際、作業しているローカルサーバ上に中間ファイルを作らずに、直接リモートホストにtarファイルを作成していることがポイントです(次ページの図参照)。

このような処理は、毎日や毎週など定期的に実行されるケースが多いでしょう。この際、バックアップ先のサーバのIPアドレスやファイルの保存パスなど多くのパラメータがあり、これを毎回手で打つのは面倒であり、タイプミスも招きます。このようにシェルスクリプトにしておけば、後は必要なときにスクリプトを実行すればよいだけなので便利ですし、パラメータ指定の誤りも防げます。

●リモートホストに直接tarファイルを作成



このスクリプトでは❶にて、**tarコマンド**でアーカイブを作成する際に、標準出力にtarアーカイブを出力する**-(ハイフン)オプション**を利用しています。❶からssh接続のためのコードを削除すると、次のようになります。

```
tar cvf - myapp/log | cat > /backup/myapplog.tar
```

ここでtarコマンドのオプションは、**c**(アーカイブ作成)、**v**(処理ファイル表示)、**f**(アーカイブファイルを使う)を利用し、これを-(ハイフン)を指定して標準出力へ表示しています。そのままではtarファイルの中身そのものが画面に表示されてしまい読めませんので、これをパイプで受け取り、**catコマンド**のリダイレクトでtarファイルとして出力することで、/backup/myapplog.tarとしてtarアーカイブを作成できます。

つまり、まずtarアーカイブを標準出力に表示し、これをパイプで受け取ります。シェル変数backup_serverで指定されたリモートホスト上でこのパイプの中身をcatすることで、リモートホスト上に、手元のファイル群を直接tarファイルとしてアーカイブできるということです。

注意事項

- このサンプル例とは逆に、リモートホストのtarファイルを直接手元で展開することもできます。この場合も、次のようにtarコマンドで-(ハイフン)を利用して、標準入力から展開するようにすればよいわけです。

```
ssh ${username}@${server} "cat /backup/myapplog.tar" | tar xvf -
```

- tarアーカイブはファイルをまとめるだけで、圧縮処理は行いません。gzip圧縮も行いたい場合は、tarコマンドにzオプション(gzip圧縮)を付加します。

関連項目

034 ファイル群を別ディレクトリに同期するバックアップ処理を行う

No. 036 重要なファイルをパスワード付きzipとしてアーカイブ

利用コマンド

zip

キーワード

パスワード, 暗号化, zipファイル

いつ使うか

重要な情報を含むログファイルなどをzipファイルにアーカイブする際、パスワード付きのzipファイルとして作成したいとき

実行例

```
$ ./passzip.sh
Enter password:
Verify password:
  adding: log/ (stored 0%)
  adding: log/access.log-20131203 (deflated 43%)
  adding: log/error.log (deflated 21%)
  adding: log/error.log-20131204 (deflated 66%)
  adding: log/error.log-20131203 (deflated 19%)
  adding: log/access.log-20131204 (deflated 60%)
  adding: log/access.log (deflated 39%)
```

パスワードをキーボードから入力する

スクリプト

```
#!/bin/sh

logdir="/home/user1/myapp"

cd "$logdir"

# /home/user1/myapp/logディレクトリ内のログファイルを、
# パスワード付きzipでアーカイブする。
zip -e -r log.zip log
```

解説

このスクリプトは、/home/user1/myapp/logというディレクトリに保存されているログファイルを、zipファイルとしてアーカイブするものです。このログファイルには重要な情報が含まれていると仮定し、パスワード付きzipファイルで保存しています。

このサンプル例で利用している**zipコマンド**の使い方は、次のようになります。

書式 zipコマンドの書式

zip *[オプション] <zipファイル> <対象ファイル>*

-rオプションは、ディレクトリ内を再帰的に処理します。すなわち、指定パスにサブディレクトリがある場合にはその中身も対象とします。大抵の場合には、このオプションが必要となるでしょう。また、単一のファイルをアーカイブする際に-rオプションを付けても特に問題はないため、基本的に-rオプションは常に付けて利用することが多いです。

パスワード付きのzipファイルを作成するには、**-e (Encrypt) オプション**を利用します。サンプル例では-e -rと別々に書いていますが、これは次のように-erと続けて書いてもかまいません。

```
zip -er log.zip myapp
```

-eオプション付きでzipファイルを作成すると、"Enter password: "とパスワードが聞かれるため、キーボードから入力します。再び"Verify password: "と聞かれますので、確認のため再度同じパスワードを入力します。これにより、パスワード付きzipファイルを作成することができます。

UNIXでよく使われるtar + gz形式では、アーカイブファイルにパスワードを設定することができません。一方、zipファイルはパスワードを設定できますし、Windowsなどでも広く利用されているため、PCとのファイルのやり取りによく使われます。このサンプル例で作ったパスワード付きzipファイルは、Windows上でも問題なく開くことができます。

注意事項

- FreeBSDでパスワード付きzipを展開する場合、バージョンによってはzipパッケージに付属の/usr/bin/unzipコマンドではエラーとなり展開できません。Portsのarchivers/unzipでインストールできる、/usr/local/bin/unzipコマンドを利用してください。

関連項目

038 tarアーカイブの際に一部のファイルやディレクトリを除外する

No. 037 gzipコマンドで圧縮率を大きくしたい

利用コマンド
gzip, bzip2, xz

キーワード
圧縮率, gzip形式

いつ使うか
他のプログラムと連携するため圧縮形式はgzから変えられないが、圧縮率を高めたいとき

実行例

```
$ ./gzip.sh
$ ls
archive.tar.gz  gzip.sh  log
```

スクリプト

```
#!/bin/sh

tar cf archive.tar log

# -9オプションで圧縮率を最大にする
gzip -9 archive.tar ―――――――――――――――――❶
```

解説

このスクリプトは、ディレクトリlogをtarアーカイブした後にgzip圧縮するものです。❶でgzip圧縮する際、-9オプションを付けて圧縮率を上げています。

gzip形式の圧縮ファイルはUNIX環境では昔から広く使われており、シェルスクリプトでも利用されることが多いファイル形式です。そのため、古くから動いている業務バッチなどでは、圧縮形式がgzであることを決め打ちにして動いているスクリプトもまだまだ多いでしょう。

現在はbzip2形式やxz形式などの、gzip形式より高圧縮率なファイル形式があり、広く使われています。本来ならばそのような高圧縮率のフォーマットに移行するのが望ましいのですが、古いシステムでは連携する他のプログラムの修正も必要なため、なかなか簡単に移行できないこともあります。

そのような場合でも何とかディスク領域を節約したいときに使えるのが、サンプル例の**-9オプション**です。**gzipコマンド**では、「-数値」オプションを付けることで、処理時間と圧縮率を調整することができます。この値は1から9まで段階的に指定することができ、数値が大きいほど圧縮率が高くなります。デフォルトの圧縮率は6です。

このサンプルでは、もっとも低速ですが圧縮率が高い-9オプションを指定しています。

これでディスク容量の節約ができます。次の実行例は、筆者の手元にある20MBほどのファイル"20140228.pcap" (tcpdumpしたパケットダンプファイル)を、gzip -1とgzip -9で圧縮してファイルサイズを比較したものです。圧縮後のファイルサイズは5.9MBと6.2MBとなり、300KBほどの差が出ました。-9オプションで圧縮したほうがより圧縮率が高いことがわかります。

gzipの圧縮率の違いによるファイルサイズ差

```
$ ls -lh
total 32M
-rw-r--r-- 1 ozuma ozuma 6.2M Mar  6 09:37 20140228.gzip1.pcap.gz
-rw-r--r-- 1 ozuma ozuma 5.9M Mar  6 09:37 20140228.gzip9.pcap.gz
-rw-r--r-- 1 ozuma ozuma  20M Mar  6 09:36 20140228.pcap
```

なお-9オプションを付けても、ファイルによってはコンマ数%程度しか圧縮率は変わらないこともあります。過度の期待はできませんが、ファイル数が多いときにはこの差も結構効いてきますので、-9オプションの利用を検討してみましょう。

注意事項

- gz形式のファイルにこだわらないならば、もっと圧縮率の高いコマンドを使うとよいでしょう。Linuxでは**bzip2コマンド**が、FreeBSDでは**xzコマンド**がよく使われます。どちらもオプションなどはgzipコマンドと互換性を持つように作られており、違和感なく使えるはずです。
- tarコマンドでアーカイブしてから圧縮する際、次のように中間ファイルを作らずに圧縮できます。こうすれば.tarのファイルをディスクに保存しなくて済むため、作業時のディスク使用量を節約できます。このtarコマンドの使い方は、P.64を参照してください。

```
tar cf - log | gzip -9 -c > archive.tar.gz
```

- gzipコマンドは、環境変数GZIPを設定すると、その値がデフォルトのオプションとして使われます。そこでスクリプトの先頭に以下のような行を入れると、常に-9オプションが付加されます。

```
GZIP='-9'; export GZIP
```

関連項目

035 ローカルディスクに実ファイルを作らず、直接リモートホストにアーカイブする
038 tarアーカイブの際に一部のファイルやディレクトリを除外する

No. 038 tarアーカイブの際に一部のファイルやディレクトリを除外する

利用コマンド
tar

キーワード
tarアーカイブ, 除外, 例外

いつ使うか
tarコマンドでアーカイブファイルを作る際、Subversionの[.svn]ディレクトリなど、特定のファイル・ディレクトリを除外したいとき

実行例

```
$ ls -aF myapp
./      ../     .svn/   bin/    etc/    log/
$ ./tar-exclude.sh
myapp/
myapp/etc/
myapp/etc/app.conf
myapp/etc/disk.conf
myapp/log/
myapp/log/access.log
myapp/bin/
myapp/bin/start
myapp/bin/stop
```

スクリプト

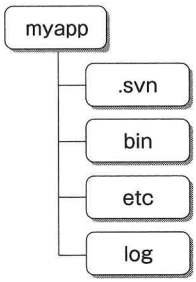
```
#!/bin/sh

tar cvf archive.tar --exclude ".svn" myapp
```

解説

このスクリプトは、tarコマンドでカレントディレクトリ配下にある[myapp]というディレクトリをアーカイブする際、[.svn]というサブディレクトリを除外するものです。このような処理はバックアップ用途などでよく使われ、定期的に実行されるものです。そのためコマンド1行で済むものであっても、ミスタイプなどを防ぐために毎回手でコマンドを打つのは避けて、このようにスクリプトを用意しておくのがよいでしょう。

ここでmyappディレクトリ内には、次のように4つのディレクトリがあると仮定しています。普通にtarコマンドを使うとmyappディレクトリ内のサブディレクトリはすべて対象となってしまいますが、ここでは[.svn]ディレクトリをアーカイブの対象外としています。そのため実行例のtarコマンドの出力に、[.svn]ディレクトリが入っていないことに注意してください。

実行例のディレクトリ構成



このようなケースでは、**tarコマンド**の**--excludeオプション**を利用することにより、指定したディレクトリを除外してアーカイブすることができます。--excludeオプションは次のように、--excludeの後にファイル名(ディレクトリ名)を指定します。

```
tar cvf archive.tar --exclude ".svn" myapp
```

なお、exclude指定した場合は、該当する名前のファイル・ディレクトリがすべて除外対象となってしまいます。例えば、「myapp/log/ディレクトリはアーカイブ対象としたいが、myapp/bakcup/log/ディレクトリは除外したい」場合は、単に"log"と指定すると両方のディレクトリが除外されてしまいます。このような場合は、ディレクトリをパス付きでexclude指定すれば除外対象を個別に設定できます。

```
tar cvf archive.tar --exclude "myapp/backup/log" myapp
```

注意事項

- アーカイブ除外したいファイルがたくさんある場合には、-Xオプションにより外部ファイルに記述した除外リストを適用することができます。例えば除外したいディレクトリを記述したテキストファイルをexclude.lstとすると、次のようになります。

```
tar cvf archive.tar -X exclude.lst myapp
```

関連項目

048 .svn などの隠しファイル・ディレクトリのみを列挙する

No. 039 tarアーカイブに後からファイルを追加する

利用コマンド
tar, date

キーワード
tar, アーカイブ, 追加, アペンド

いつ使うか
月次アーカイブに日次でファイルを追加するなど、既存のtarアーカイブにファイルを追加したいとき

実行例

```
$ tar tf 201312.tar ―――― 既存ファイルの確認
log/
log/20131201.log
log/20131202.log
log/20131203.log
log/20131204.log
$ ./tar-add.sh ―――― 実行
log/20131205.log
$ tar tf 201312.tar ―――― 追加されたことを確認
log/
log/20131201.log
log/20131202.log
log/20131203.log
log/20131204.log
log/20131205.log
```

スクリプト

```
#!/bin/sh

# 年月でアーカイブファイルを指定 (例:201312.tar)
archivefile="$(date +'%Y%m').tar" ―――― ❶
# 今日の日付からログファイルを指定 (例:20131205.log)
logfile="$(date +'%Y%m%d').log"

# 月次アーカイブに、今日のログを追加
tar rvf $archivefile log/$logfile ―――― ❷
```

解説

このスクリプトは、月次で作られるtarアーカイブファイルに、毎日作られる日付名のログファイルを追加していくものです。ここでは今日は2013年12月5日であると仮定し、

201312.tarというアーカイブファイルに20131205.logというログファイルを追加するケースを想定しています。

tarコマンドでは、cオプションとfオプションを利用することで、新規アーカイブを作成することができますが、すでに存在するtarアーカイブファイルの最後にファイルを追加することもできます。この使用例が、❷で指定している**rオプション**(appendオプション)です。

実行例ではまずはじめに、tarコマンドのtfオプションでアーカイブ内を確認しています。結果を見るとわかるように、このサンプル例では、はじめは201312.tarには以下の4ファイルがアーカイブされていました。

・20131201.log
・20131202.log
・20131203.log
・20131204.log

ここでサンプルスクリプトを実行すると、まず**dateコマンド**を利用して、現在の日付をもとに月次アーカイブファイルとログファイルのファイル名を組み立てます(❶)。この、dateコマンドを利用して日付でファイル名を組み立てる方法については、P.88で解説していますので詳しくはそちらを参照してください。

ファイル名が組み立てられたら、❷にてtarコマンドでアーカイブファイルに今日の日付のログファイルを追加しています。rオプションを利用しているため、前日ぶんまでのファイルはそのままに、今日のファイルを追加することができます。こうして月次ごとのtarアーカイブを自動で作ることができます。

注意事項

・**rオプションを付けた際、指定したtarアーカイブファイルが存在しない場合は、新規にtarアーカイブファイルが作成されます。エラーとはなりませんので注意してください。**

関連項目

033 ファイルをバックアップする際にファイル名に日時を入れる
038 tarアーカイブの際に一部のファイルやディレクトリを除外する

No. 040 ファイルパーミッションやタイムスタンプなど、元のファイルの属性を保ったままファイルコピーをする

利用コマンド
getopts, cp

キーワード
コピー , バックアップ, ファイル属性, シンボリックリンク

いつ使うか
ディレクトリのコピーをとる際、ファイル属性やシンボリックリンクを含めてバックアップしたいとき

実行例

```
$ ./cp-p.sh -a
```

スクリプト

```
#!/bin/sh

backup_dir="/home/user1/backup"

# myappディレクトリを、$backup_dir下にバックアップコピー
while getopts "a" option
do
  case $option in
    a)
      cp -a myapp "$backup_dir"
      exit
      ;;
  esac
done

cp -R myapp "$backup_dir"
```

解説

このスクリプトは、カレントディレクトリにあるmyappというディレクトリ配下のファイル・ディレクトリ一式を、シェル変数backup_dirで指定されたディレクトリへとコピーして、バックアップを取得するものです。この際、-aオプションを指定すると、ファイルのタイムスタンプやパーミッションなどの、ファイル属性を保持してコピーを行います。

サンプル例では、**getoptsコマンド**を利用してシェルスクリプトへのオプション指定を判断しています。getoptsの使い方について詳しくはP.2を参照してください。ここでは-a

を指定されると**cpコマンド**を**-aオプション**付きで実行してタイムスタンプやパーミッションを保持してコピーします。一方、何もオプション指定されないとcpコマンドを-Rオプション付きで実行して単にファイルコピーだけを行うという動きにしてみました。この後者の動きは、例えばコピーした日時を明確にしたい場合には、タイムスタンプをあえて保持せずにコピーするケースを想定しています。

cpコマンドでは、何もオプションを付けずにファイルをコピーすると、ファイルのパーミッションはumaskで設定された値に変わります。また、タイムスタンプは現在の時刻で更新されてしまいます。加えて、オプションを付けないとcpコマンドはディレクトリのコピーをしてくれません。バックアップ用途でディレクトリ一式などをコピーする際には、これでは困る場合があります。

-aオプションを付けると、cpコマンドはオリジナルファイルの所有者・グループ・アクセス権・アクセス時刻などのファイル属性を保持したままファイルをコピーします。そのため、バックアップ用途でよく使われるオプションです。また、-aオプションを利用すると、**-R (recursive) オプション**も同時に指定されたとみなされ、サブディレクトリを含めてファイルツリーをそのままコピーしてくれます。

-Rオプションを付けると、シンボリックリンクはそのリンク先を追わず、リンクそのものとしてコピーされます。もしシンボリックリンクが指している先の実体ファイルとしてコピーしたい場合は、次のように-Rと**-Lオプション**を同時に利用します。また、**-p (preserve) オプション**を付けるとファイル属性も保持できます。

```
cp -pRL myapp "$backup_dir"
```

なおcpコマンドのmanを読むとわかりますが、cpコマンドの-aオプションは、-pオプションにいくつかのオプションを組み合わせて、オプション指定を簡略化するために用意されているものです。この-aオプションはOSによって、次のように多少動作が異なります。

- Linuxの場合、-aは-dpRと同じ。-dはシンボリックリンクをシンボリックリンクとしてそのままコピーすることを意味する。
- FreeBSD/Macの場合、-aは-RpPと同じ。-Pは、シンボリックリンクをシンボリックリンクとしてそのままコピーすることを意味する。

バックアップスクリプトとしては、-aオプションを使わずに-pオプションと-Rオプションを組み合わせる例も一般的によく用いられています。また同様の例として、次のように小文字の-rで書かれたスクリプトも見ることが多いかもしれません。

```
cp -pr myapp "$backup_dir"
```

しかし小文字の**-rオプション**は「方言」があり、例えばLinuxのcpコマンドでは-rでコピーするとシンボリックリンクをそのままリンクとしてコピーしますが、MacやFreeBSD

のcpコマンドで-rオプションを利用するとシンボリックリンクが指している実体ファイルをコピーします。

そのため、特にBSD系のシステムでは、cpコマンドの-rオプションは使うべきではなく、-Rを利用することが推奨されています。興味のある方は、FreeBSDのcpコマンドのmanを読んでみてください。

注意事項

- コピー元のファイルについて、スクリプト実行者とファイル所有者が違う場合には属性を保持できない場合があります。例えば、オーナーがrootユーザのファイルを一般ユーザが-pオプション付きのcpコマンドでコピーしても、ファイルの所有者をrootとする操作は一般ユーザには許可されていないため、ファイルの所有者属性はcpコマンドの実行者になります。ただし、ファイルのタイムスタンプなどはそのまま維持できます。

rootが所有するファイルを一般ユーザが-aオプションでコピーしても･･･

```
$ ls -l
total 0
-rw-r--r-- 1 root root 0 Dec  6 22:55 test.txt
$ cp -a test.txt my.txt
$ ls -l
total 0
-rw-r--r-- 1 user1 user1 0 Dec  6 22:55 my.txt
-rw-r--r-- 1 root  root  0 Dec  6 22:55 test.txt
```

-aオプション付きのcpコマンドでコピーしたが、my.txtの所有者はrootではなくuser1のものになった

- 同様に、ファイルのグループ属性が自分が所属していないグループとして設定されている場合は、root権限でなければグループ属性をそのままコピーすることはできません。
- このサンプル例と同様の手法として、中間ファイルを作らずに対象ファイルをtarアーカイブして、コピー先に展開する方法があります。P.94を参照してください。

関連項目

035 ローカルディスクに実ファイルを作らず、直接リモートホストにアーカイブする

No. 041 拡張子に.htmと.htmlが混じったHTMLファイル群の拡張子を一括してtxtに変更する

利用コマンド
mv

キーワード
拡張子, リネーム, case文

いつ使うか
拡張子が入り交じったファイル群を、1つの拡張子にまとめてリネームしたいとき

実行例

```
$ ls
abc.html  index.htm  rename.sh  same.htm  same.html  sp ace.html
$ ./rename.sh
$ ls
abc.txt  index.txt  rename.sh  same.txt  sp ace.txt
```

スクリプト

```
#!/bin/sh

for filename in *                                    ❶
do
  case "$filename" in                                ❷
    *.htm | *.html)                                  ❸
      # ファイル名の前の部分を取得（index）
      headname=${filename%.*}                        ❹

      # ファイル名を.txtに変換
      mv "$filename" "${headname}.txt"               ❺
    ;;
  esac
done
```

解説

このスクリプトは、カレントディレクトリにある拡張子.htmおよび.htmlのファイルすべてについて、拡張子を.txtに変更するものです。

まず❶にて、*を利用して、カレントディレクトリのファイルリストに対してシェル変数filenameを用いて順に処理していきます。このようにfor文のinに*を指定することでパス名展開され、カレントディレクトリのファイルリストを簡単に作成することができます。

❷では、ファイル名についてパターンマッチを行います。ファイル名から指定文字列を

含むものだけを選んで処理する場合には、このように**case文**を利用する方法が簡便でよく使われます。ここでは拡張子.htmもしくは.htmlの場合にリネーム処理を行いたいため、マッチするパターンの指定としては「任意の文字列＋.htm」もしくは「任意の文字列＋.html」となります。これはすなわち「*.htm | *.html」となり、❸にこのパターンを記述しています。

❹は、**パラメータ展開**を用いてファイル名から「拡張子を除いたファイル名」を取り出す際によく使われる書き方で、慣用句のようなものです。指定されたシェル変数から、%.*という記法で、「.（ドット）＋任意の文字列」を削除することで拡張子を除いた部分を取得できますので、これをシェル変数headnameに代入しています。この%を用いた記法については、P.62の注意事項で紹介していますので詳しくはそちらを参照してください。

最後に❺で、拡張子を.txtに変更します。なお**mvコマンド**の2つの引数をダブルクォートでくくっているのは、空白文字（スペース）を含むファイル名にも対応するためです。

注意事項

- このスクリプトでは、拡張子だけが違うファイルが2つあった場合、片方の内容で.txtファイルは上書きされてしまいます。

片方のファイルで上書きされてabc.txtが1つだけとなる

```
$ ls
abc.htm  abc.html
$ ./rename.sh
abc.txt
```

- Linuxには**rename**というコマンドがあり、ファイル名の一部を簡単に変更することができます。このコマンドを利用すれば、サンプル例と同じ動作を次のように短く書くことができます。

```
rename .htm .txt *.htm
rename .html .txt *.html
```

関連項目

023 絶対パスで起動されても相対パスで起動されても、同じ動作をできるようにする
024 コマンドの使い方を表示する際に、現在の自分自身のファイル名を使って例示する

No. 042 処理開始前に、実行権限をチェックして正常動作できることを確認してから実行する

利用コマンド
test, [

キーワード
ファイル属性, 状態チェック, ファイルテスト, パーミッション

いつ使うか
スクリプトの初期設定などで、特定のファイルの存在チェックやパーミッションチェックを行いたいとき

実行例

```
$ ./test.sh
start.
```

スクリプト

```
#!/bin/sh

start_command="./start.sh"

if [ -x "$start_command" ]; then
  $start_command
else
  echo "ERROR: -x $start_command failed." >&2
  exit 1
fi
```

❶

解説

このスクリプトは、カレントディレクトリにあるstart.shを実行するだけの簡単なものです。start.shを実行できるかどうかを、スクリプト内で事前にパーミッションチェックをして判断しています。なお、ここでstart.shとは何らかの処理を行うスクリプトであると仮定しており、このサンプル例では"start."と表示するだけの単純なスクリプトです。

このサンプル例では、スクリプト内で別のコマンドが実行できるかを、**testコマンド**の、ファイルの実行権限を調べる演算子**-x**でチェックしています。ファイルの状態を調べる演算子はたくさんありますが、次ページの表ではよく使われるものをあげておきました。例えばスクリプトの起動時の初期チェックとして、ログディレクトリに書き込みできるかどうかを-wで調べたり、設定ファイルが読めるかを-rでチェックしたりという用途でよく使われます。

testコマンドの主な演算子

指定	説明
-d	ファイルが存在し、ディレクトリなら真
-e	ファイルが存在すれば真
-b	ファイルが存在し、ブロックスペシャルファイルであれば真
-c	ファイルが存在し、キャラクタスペシャルファイルであれば真
-f	ファイルが存在し、通常ファイル (regular file) であれば真
-L	ファイルが存在し、シンボリックリンクであれば真
-r	ファイルが存在し、読み取り可能であれば真
-w	ファイルが存在し、書き込み可能であれば真
-x	ファイルが存在し、実行可能であれば真
-s	ファイルが存在し、サイズが0より大きければ真
-O	ファイルが存在し、実行中のシェルの実行ユーザIDに所有されていれば真
-S	ファイルが存在し、ソケットファイルであれば真

これら演算子はまだまだたくさんありますので、もっと詳しく知りたい方は、man testとしてtestコマンドのマニュアルを読んでみてください。

なおここまで、testコマンドという記述をしてきました。実は❶のif文の中に表れるカッコ [は、制御構造を示すカッコではなく、testコマンドと同じくコマンドです。例えば次のようにコマンド名を解釈するtypeコマンドで調べてみると、**[コマンド**はシェルビルトインのコマンドとして用意されているのがわかると思います。

[がコマンドであることを確認する

```
$ type [
[ is a shell builtin
```

ただし、[コマンドがtestコマンドと違う点として、**[コマンドは最後の引数に閉じカッコ]を必要とする**点があげられます。このため、if文の条件式として使うと、条件全体がカッコでくくられて見やすくなるため好んで使われます。つまりサンプルの❶は、次のように書くことと実は同じです。

```
if test -x "$start_command"; then
```

testコマンドは、条件判断をしてその結果が真ならば終了ステータスにゼロを返します。if文ではこの終了ステータスに基づいて、真偽値を判断しているわけです。

一般的には、if文の条件式として[]を使うことが多いですが、testコマンドで書いてもかまいませんし、あるいは他にも終了ステータスを判断したいコマンドを何でも書くことができます。例えばリスト1は、if文の条件式にgrepコマンドをそのまま書いています。ここではファイルsample.txtに、文字列"bin"が含まれているかを調べています。

リスト1 if文の条件式にgrepコマンドをそのまま書く

```
#!/bin/sh

if grep -q "bin" sample.txt ; then
  echo "sample.txt has string [bin]"
fi
```

grepコマンドは、マッチする文字列があった場合には終了ステータス0を、マッチしなかった場合は非ゼロである1を返しますから、このようにif文の条件式としてそのまま使うことができるわけです。

注意事項

- サンプルであげた「もしファイルが実行可能ならば実行する」という単純な条件を簡単に書くために、次のような記法もよく用いられます。

```
test -x start.sh && ./start.sh
```

&&でコマンドをつなげると、前のコマンドが成功したら(すなわち終了ステータスが0ならば)次のコマンドを実行する、という形で書くことができます。ここではtest -xコマンドが成功すればファイルが実行可能なことを意味するので、続いてそのコマンドを実行するよう記述しているわけです。

- このサンプル例では、シェル変数start_commandで指定されたファイルがディレクトリであるかどうかは調べていません。そのため、あまりないケースですが、[start.sh]という名前のディレクトリがあるとエラーになります。厳密にやるならば、❶のif文の中身を次のように書き、-fで通常ファイルかどうかも調べてから実行するとよいでしょう。この際、**-a**というのはtestコマンドで「AND」を意味する演算子です。

```
if [ -f "$start_command" -a -x "$start_command" ]; then
```

関連項目

043 2つのファイルの新旧を比較し、古いほうを削除する

No. 043 2つのファイルの新旧を比較し、古いほうを削除する

利用コマンド
test, rm

キーワード
タイムスタンプ, 新旧

いつ使うか
タイムスタンプを比較し、どちらのほうが古いかを判定したいとき

実行例

```
$ ./olddel.sh
[log2.log]->newer, [log1.log]->older
```

スクリプト

```
#!/bin/sh

# 新旧を比較する対象ファイル
log1="log1.log"
log2="log2.log"

# 引数のファイルが存在するかを調べ、存在しない場合は終了する
filecheck() ──────────────────────────────❶
{
  if [ ! -e "$1" ]; the ──────────────────❷
    echo "ERROR: File $1 does not exist." >&2
    exit 1;
  fi
}

filecheck "$log1"
filecheck "$log2"  ──────────────────────────❸

# 2つのファイルの新旧を比べ、古いほうを削除する
if [ "$log1" -nt "$log2" ]; then
  echo "[$log1]->newer, [$log2]->older"
  rm $log2
else                                        ❹
  echo "[$log2]->newer, [$log1]->older"
  rm $log1
fi
```

解説

このスクリプトは、シェル変数log1とlog2で指定された2つのファイルのタイムスタンプ(modify time)を調べ、そのうち古いほうのファイルをrmコマンドで削除するものです。

まずはじめに、タイムスタンプを比較するファイルが存在するかどうかのチェックを行うためにfilecheckというシェル関数を用意しました(❶)。この中では、if文と**testコマンド**の**-e**によってファイルが存在するかのチェックを行っています。このようなファイルテストについて、詳しくはP.109を参照してください。

なお❷では、ファイルの存在を-eでチェックする際に!という否定演算子を付けて、全体の真偽を逆にしています。つまり、-eで「ファイルが存在する」場合に真になるので、それを否定することで、「ファイルが存在しない場合」の処理を書いていることになります。ここではファイルが存在しない場合、エラーメッセージを表示して**exitコマンド**でスクリプトを終了しています。

また❷において利用している$1という変数は、**位置パラメータ**(→P.4)と呼ばれます。これはシェル関数の中では関数への引数を表します。ここでは❸でチェックしたいログファイルを指定するシェル変数を渡していますから、"log1.log"などのファイル名が代入されていることになります。

ファイルのタイムスタンプの新旧を比較しているのが、❹のif文です。**-nt**はnewer thanの略で、次のように利用してファイルのタイムスタンプを比較することができます。

```
if [ <ファイル1> -nt <ファイル2> ]; then
```

つまりこのif文が真の場合にはファイル2のほうが古く、偽の場合はファイル1のほうが古いということです。この結果から、サンプル例では**rmコマンド**を使うことによって古いほうのファイルを削除しています。

注意事項

- もし2つのファイルのタイムスタンプが全く同一の場合、-ntは「より新しい」かを調べるため偽を返します。つまりこのサンプル例ではlog1のほうが古いとみなされ削除されることになります。
- **-ot** (older than)という条件式もあります。これは-ntとは逆で、より古いファイルの場合に真となります。

関連項目

028 新規ファイルを作らずに、すでにあるファイルのみファイル更新日を変更する
042 処理開始前に、実行権限をチェックして正常動作できることを確認してから実行する

No. 044 2つのディレクトリ内を比較し、どちらか片方だけに存在するファイルを表示する

利用コマンド
find, sort, comm

キーワード
ファイルリスト, ディレクトリ比較

いつ使うか
2つのディレクトリが似たような構成の際に、片方だけにある・両方にあるファイルを見やすくリストアップしたいとき

実行例

```
$ ./find-comm.sh
./dav.conf
                ./default.conf
                ./info.conf
                ./mpm.conf
        ./ssl.conf
                ./userdir.conf
./vhosts.conf
```

スクリプト

```
#!/bin/sh

# 比べる2つのディレクトリ名
dirA="dir1"
dirB="dir2"

# dir1/ と dir2/ のファイルリストの差を調べる。
( cd ${dirA}; find . -maxdepth 1 -type f -print | sort ) > tempfile1.lst ❶
( cd ${dirB}; find . -maxdepth 1 -type f -print | sort ) > tempfile2.lst ❶

comm tempfile1.lst tempfile2.lst ❷
```

解説

このスクリプトは、カレントディレクトリ配下の2つのサブディレクトリ、dir1とdir2の中のファイルを調べ、

(1) dir1のみに存在するファイル
(2) dir2のみに存在するファイル
(3) dir1とdir2両方に存在するファイル

を分けて出力するものです。出力としては、タブ区切りで左から(1)、(2)、(3)の順に表示されます。つまりこの実行例では、dav.confとvhosts.confがディレクトリdir1のみに存在し、ssl.confがディレクトリdir2のみに存在し、残りのファイルは両方のディレクトリに存在する、ということになります。読者がこのサンプル例を試す際には、最初に定義しているシェル変数dirAとdirBの値を、実際に比べたい2つのディレクトリ名に変更して実行してください。

❶で、それぞれのサブディレクトリの中のファイルリストを一時ファイルとして出力しています。ここでは全体を**サブシェル**(→P.66)として、カレントディレクトリの移動やリダイレクト出力を行っています。サブシェルとしているのは、**cdコマンド**でカレントディレクトリを移動した際に、元のディレクトリに自動的に戻るようにするためです。

また❶では、ディレクトリ内のファイルリスト作成の際に、**findコマンド**で**-type f**と指定することによりディレクトリを除外し、通常のファイルのみを指定しています。また**-maxdepth 1**として、2階層以上深いディレクトリ内のファイルは対象とせず、サブディレクトリ直下のファイルのみを対象としています。findコマンドの基本的な使い方は、P.68を参照してください。

ここまでで、ディレクトリdir1の中のファイルリストをtempfile1.lst、ディレクトリdir2の中のファイルリストをtempfile2.lstとして出力できたので、この2つのファイルリストを比較します。

❷で、**commコマンド**を使って2つのディレクトリ内のファイルリスト差分を出力しています。commコマンドとはファイルの内容を比較するコマンドで、2つの入力ファイルを読み込み、共通な行および共通でない行をそれぞれ表示するコマンドです。commコマンドの実行例は次のようになり、3列からなる出力を生成します。

commコマンドは2つのファイルを比較する

```
$ comm file1 file2
                    1
2
          3
                    4
                    5
                    6
7
```

ここで、それぞれの列の意味は以下のようになります。列はタブ区切りとなります。

・第1列にはfile1だけに含まれる行を出力

・第2列にはfile2だけに含まれる行を出力
・第3列には両方のファイルに共通に含まれている行を出力

なおcommコマンドを利用する際は、入力ファイルがソートされている必要があります。そのためこのサンプル例では、❶でfindを実行後に**sortコマンド**で行ソートしてから一時ファイルに出力しています。こうしてディレクトリ内のファイルリストをcommコマンドに比較させることで、各ディレクトリ内のファイルのチェックが行えます。

注意事項

・このように2つのディレクトリ間でファイルリストの差分を確認した後には、ディレクトリ間でファイルを同期させたくなることでしょう。2つのディレクトリの中のファイルを同期させたいときには、rsyncコマンド(→P.91)が便利です。
・commコマンドは、指定した列のみの表示をすることができます。この際は、表示したくない列を-1あるいは-2のように指定します。例えば両方のファイルに存在する行のみ表示したいときは、次のように-12を指定して第3列のみ表示させればよいわけです。

```
comm -12 file1 file2
```

・commコマンドの終了ステータスは、似た動きをする**diffコマンド**と若干異なるため注意が必要です。diffコマンドの終了ステータスは以下のようになっています。

2つのファイルが同一ファイルのときは終了ステータス0
2つのファイルに差があるときは終了ステータス1
ファイルが見つからないなどのエラー時には終了ステータス2

つまりdiffコマンドでは、終了ステータスが0か1かで、2つのファイルに差があるかどうかを判断するのが一般的です。
一方、commコマンドは2つのファイルに差があってもなくても、正常終了すれば終了ステータスに0を返します。commコマンドが終了ステータスに非ゼロを返すのは、ファイルが見つからないなどのエラー時だけです。このため、commコマンドの終了ステータスをチェックして条件分岐をするスクリプトを作る際には、diffコマンドとの違いに注意してください。

関連項目

025 ディレクトリ移動した後に簡単に元の場所に戻る
026 ディレクトリ内のファイル数・ディレクトリ数を調べる
034 ファイル群を別ディレクトリに同期するバックアップ処理を行う

No. 045 あるディレクトリの中で容量を食っているサブディレクトリを調べる

利用コマンド
du, sort

キーワード
ディスク使用量, ファイルサイズ

いつ使うか
ディレクトリごとのディスク使用量を表示したいとき

実行例

```
$ ./du-sub.sh
29116    /home/user1/myapp/data/dir1/
4716     /home/user1/myapp/data/dir2/
1020     /home/user1/myapp/data/dir3/
```

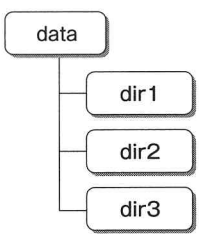
スクリプト

```
#!/bin/sh

data_dir="/home/user1/myapp/data"

# ディレクトリ$data_dir配下の、サブディレクトリの
# 容量をサマリ表示する
du -sk ${data_dir}/*/ | sort -rn ―――――――――――――❶
```

解説

このスクリプトは、シェル変数data_dirで指定されたディレクトリの中の、サブディレクトリごとのディスク使用量を調べるものです。ここで、dataディレクトリの中にはdir1、dir2、dir3という3つのディレクトリがあり、その中にはたくさんのファイル・ディレクトリが含まれているものとします。

実行例のディレクトリ構成



ファイルが利用しているディスク使用量を調べるには、**duコマンド**を用います。これは意外に扱いにくくわかりにくいコマンドですので、ここで使い方を少し詳しく解説します。

まずはじめに、duコマンドで表示される値について説明しておきましょう。duコマンドで表示される値は、ファイルサイズそのものではなく、ファイルがディスク上で利用しているブロックサイズを示します。このブロックサイズは環境によって値が変わりますので、シェルスクリプト内でduコマンドを利用する際は、常に**-kオプション**(キロバイト単位)や**-mオプション**(メガバイト単位)で表示するよう指定したほうがよいでしょう。例えば次の環境ではブロックサイズが512バイトであるため、-kオプションを付けて1024バイト単位の表示とすると、報告される値が半分になります。

-kオプションを付けるとキロバイト単位で表示する

```
$ du
1544    .
$ du -k
772     .
```

さて、duコマンドを引数なしで実行すると、カレントディレクトリ配下のディレクトリすべてについて、それぞれのディレクトリが利用しているディスク使用量を表示します。

duコマンドを引数なしで実行するとサブディレクトリも表示する

```
$ cd /home/user1/myapp/data
$ du
29116   ./dir1
8       ./dir3/controller
8       ./dir3/agent
1020    ./dir3
4716    ./dir2
35856   .
```

上記では、dir3というディレクトリの中にはさらにcontrollerとagentというサブディレクトリがあるため結果に表示されています。しかし一般的にこういう表示はなかなか見づらく、dir3でまとめた使用量を表示してほしいと思うでしょう。本サンプル例は、そのような用途のために使うものです。

リストの❶では、duコマンドに**-sオプション**を利用しています。これは実際に引数で指定された対象のディスク使用量のみを表示し、そのサブディレクトリによる使用量は表示しないオプションです。❶では、-sオプションに/*/という引数を与えることで、サブディレクトリごとの容量を表示しています。また同時に-kオプションを指定して、キロバイト単位で表示しています。

続いて❶では、duコマンドの出力結果をパイプで**sortコマンド**に渡し、**-r**で逆順ソートしています。これでディスク使用量の多い順に表示することができます。またこの際、**-nオプション**も同時に利用して、文字列ソートではなく数値ソートとなるようにしています。-nオプションを利用しないと文字列ソートとなってしまうため、このような数値の降順ソートを行いたい場合は、-rn指定するのがよく使われるやり方です。

duコマンドの-hオプション

duコマンドは**-h (human-readable) オプション**を利用すると、キロバイト(K)やメガバイト(M)など単位付きで使用量を読みやすく表示できます。

大きなバイト数は-hオプションで読みやすくなる

```
$ du -h
29M     ./data/dir1
8.0K    ./data/dir3/controller
8.0K    ./data/dir3/agent
1020K   ./data/dir3
4.7M    ./data/dir2
36M     ./data
36M     .
```

このように-hオプションを付けると大変読みやすくなりますが、sortコマンドで使用量ごとにソートすることができなくなってしまいます。そのため、目で見て確認するときには便利ですが、シェルスクリプト中でソートしたいときにはあまり使いません。

なお、一部のLinuxのsortコマンドには-h (--human-numeric-sort) オプションがあり、2K、1G、というようなduコマンドの-hオプションでの出力値を正しくソートできます。

関連項目

116 ディスクの容量監視を行う

No. 046 作業ファイルを作る際に、内容を読まれないようセキュリティ対策を行う

利用コマンド
umask

キーワード
セキュリティ , パーミッション, umask

いつ使うか
スクリプト内で、パーミッションを指定してファイルを作成したいとき

実行例

```
$ ./umask.sh
$ ls -l
total 8
-rw------- 1 user1 user1 48 Dec 11 23:24 idinfo.tmp
-rwxr-xr-x 1 user1 user1 39 Dec 11 23:17 umask.sh
```

スクリプト

```
#!/bin/sh

umask 077 ―――――――――――――――――――――――――――――❶

# echoコマンドの出力を、一時ファイルとしてパーミッション600で作成
echo "ID: abcd123456" > idinfo.tmp ―――――❷
```

解説

このスクリプトは、umaskコマンドを用いて、一時ファイルのパーミッションを設定しておくものです。こうして他の人からファイル内容を読まれないようにセキュリティに配慮して、ファイルを作成することができます。

サンプル例では、"ID: abcd123456"という文字列を一時ファイルに出力しています。

スクリプトの実行中に一時ファイルを作ることはよくありますが、このようにID情報など、あまり他の人に読まれたくない情報を記録することがあるでしょう。このような場合は、**umaskコマンド**でマスク値を設定してからファイルを作れば、一時ファイルのパーミッションを適切に設定できます。

umaskコマンドの使い方は、次のようになります。

書 式 umaskコマンドの書式

umask *<マスク値>*

マスク値(umask値)は3桁の8進数で指定します。umaskコマンド実行後に作られるファイルのパーミッションは、umaskで指定したビットが0になるように作成されます。また、ファイルのパーミッションの値は、一般的なシェルスクリプト上でファイルを作る際は、666(ディレクトリの場合は777)をumask値でマスクした値になります。

つまり666とumask値それぞれを2進数で記述し、umaskの値が1になっている桁を0とした値が、スクリプト中で作成されるファイルのパーミッションになります。

例えば、umask 022と指定したときに作られるファイルのパーミッションは、次のとおり644となります。

パーミッションの指定方法

```
666  --2進数表示--> 1 1 0 1 1 0 1 1 0
022  --2進数表示--> 0 0 0 0 1 0 0 1 0

      マスク結果    1 1 0 1 0 0 1 0 0 → 8進数になおすと[644]
```

このスクリプトでは、スクリプトの先頭(❶)で、umask値に077を設定しています。これにより、このスクリプト内で作られるファイルのパーミッションは600(ファイル所有者のみ読み書き可)となります。❷でechoコマンドの出力をファイルにリダイレクトしていますが、この一時ファイルのパーミッションも600となり、他の人に読まれる心配がありません。こうしてシェルスクリプト中で、安全に一時ファイルを作成することができるということです。

関連項目

049 二重起動が可能な一時ファイルを作成する

No. 047 バイナリファイルに含まれる文字列を取得する

利用コマンド
strings, grep

キーワード
バイナリファイル, 検索, 実行ファイル

いつ使うか
エラーメッセージを出しているコマンドが不明な環境で、そのコマンドを探したいとき

実行例

```
$ ./strings.sh
/home/user1/myapp/bin/start: error: Unknown Error
/home/user1/myapp/bin/kill: Unknown Error at %s
/home/user1/myapp/bin/kill: Unknown Error at %s:%d
```

スクリプト

```
#!/bin/sh

# 検索したいエラーメッセージ
message="Unknown Error"

strings -f /home/user1/myapp/bin/* | grep "$message"  ❶
```

解説

このスクリプトは、あるエラーメッセージを手がかりに、バイナリファイルの中からそのエラーメッセージを出力しているコマンドを探すものです。

システム運用をしていると、何かエラーメッセージが出ているのに、それをどのコマンドが出しているのかがわからない、という事態が起きることがあります。プログラムがPerlやRubyなどのスクリプト言語で書かれているならば、そのエラーメッセージで単純にプログラムファイルをgrepしてみればよいでしょう。しかしC言語などで書かれていると、コンパイルされた実行ファイルはバイナリファイルのため、単純にテキストとして検索することができません。

こんなときに、エラーメッセージを表示しているコマンドを探す手段の1つとして使えるのが**stringsコマンド**です。

stringsコマンドは、バイナリファイルの中から文字列を抽出してくれるコマンドです。一般的に、C言語で書かれたコンパイル済みのバイナリファイルであっても、多くの場合は文字列定数はファイル中にそのまま格納されています。そのため、stringsコマンドでプログラム内に書かれたエラーメッセージを見つけ出すことができます。このサンプル例は、そうしてエラーメッセージを出力しているコマンドの「当たり」を付ける…という、

トラブルシューティング時に使えるスクリプトです。

リストの❶では、stringsコマンドに**-fオプション**を付けて、文字列表示の際にファイル名も同時に表示するようにしています。-fオプションは対象ファイルをワイルドカード(*)で指定することにより、指定したパス内のファイルすべてを対象にしています。stringsコマンドの出力は、パイプで**grepコマンド**に渡して、シェル変数messageで指定した文字列にマッチする場合だけ表示しています。このサンプル例では、結果として"Unknown Error"という文字列を含むファイルを抽出できます。

このサンプル例は、例えば読者が運用しているアプリケーションが、ログファイルに、見たことのない不審なメッセージを出力しているケースで利用できます。ログファイルの不審なメッセージをシェル変数messageに設定し、stringsコマンドの-fオプションで指定するパスをアプリケーションのディレクトリに設定して実行してみてください。

odコマンドとhexdumpコマンド

バイナリファイルの中身を直接見るには、ファイルを8進数でダンプする**odコマンド**がよく使われます。odコマンドはさまざまなオプションを持ちますが、単にバイナリファイルの中にある文字列を見たいだけならば、ASCII文字出力をする**-cオプション**だけ覚えておけば十分です。

次は、Linuxのカーネルファイルをodコマンドでダンプした実行例です。"Direct booting from..."という文字列があるのがわかります。

Linuxカーネルをodコマンドでダンプ

```
$ od -c vmlinuz-2.6.32-358.23.2.el6.x86_64
0000000 352 005  ¥0 300  ¥a 214 310 216 330 216 300 216 320   1 344 373
0000020 374 276   -  ¥0 254     300   t  ¥t 264 016 273  ¥a  ¥0 315 020
0000040 353 362   1 300 315 026 315 031 352 360 377  ¥0 360   D   i   r
0000060   e   c   t       b   o   o   t   i   n   g       f   r   o   m
0000100       f   l   o   p   p   y       i   s       n   o       l   o
…(省略)
```

なおodコマンドは比較的シンプルなコマンドです。もう少し高機能なものとして、**hexdumpコマンド**もよく使われます。hexdumpコマンドは-Cオプションを利用すると、ファイルの内容を、「16進ダンプとASCII文字列のセット」で表示することができます。

Linuxカーネルをhexdumpコマンドでダンプ

```
$ hexdump -C vmlinuz-2.6.32-358.23.2.el6.x86_64
00000000  ea 05 00 c0 07 8c c8 8e  d8 8e c0 8e d0 31 e4 fb  |.............1..|
00000010  fc be 2d 00 ac 20 c0 74  09 b4 0e bb 07 00 cd 10  |..-.. .t........|
00000020  eb f2 31 c0 cd 16 cd 19  ea f0 ff 00 f0 44 69 72  |..1..........Dir|
00000030  65 63 74 20 62 6f 6f 74  69 6e 67 20 66 72 6f 6d  |ect booting from|
00000040  20 66 6c 6f 70 70 79 20  69 73 20 6e 6f 20 6c 6f  | floppy is no lo|
…（省略）
```

注意事項

- プログラム内で動的にエラーメッセージを組み立てている場合などは、stringsコマンドでは該当のバイナリファイルを見つけることはできないでしょう。
- Macのstringsコマンドには-fオプションがないため、ファイル名を表示させることができません。そのため❶の部分は、forループを使ってファイルごとに処理するとよいでしょう。

```
for filename in /usr/local/bin/*
do
  echo "$filename:"
  strings $filename | grep "$message"
done
```

No. 048 .svnなどの隠しファイル・ディレクトリのみを列挙する

利用コマンド
ls, case

キーワード
隠しファイル, ドットファイル

いつ使うか
隠しファイルのみを対象とした処理を行いたいとき

実行例

```
$ ./dotfile.sh
dot file: .bashrc
dot file: .cshrc
dot directory: .svn/
```

スクリプト

```
#!/bin/sh

# IFSに改行を設定する
IFS='
'                                          ❶

# カレントディレクトリ配下のファイルを$filenameとして順に処理
for filename in $(ls -AF)                  ❷
do
  case "$filename" in                      ❸
  .*/)
    echo "dot directory: $filename"
  ;;
  .*)
    echo "dot file: $filename"
  ;;
  esac
done
```

解説

このスクリプトは、カレントディレクトリにあるドットファイル(ディレクトリ)を列挙し、それぞれがファイルなのかディレクトリなのかを表示するものです。

一般的にUNIXでは、ファイル名の先頭が.(ドット)で始まるファイルを**ドットファイル**と呼び、これを隠しファイルとして扱います。ドットファイルはコマンドの設定など特殊な用途に使われることが多く、何かと特別扱いすべきことも多いため、このサンプル例の

ようにドットファイルのみを対象とするスクリプトが必要な場面があるでしょう。よく使われるドットファイルを、次の表にあげておきました。

よく使われるドットファイル(ディレクトリ)

名前	用途
.bash_profile	bashがログイン時に読み込む環境設定ファイル
.mysql_history	mysqlコマンドの実行履歴ファイル
.vimrc	vi (vim) エディタの環境設定ファイル
.DS_Store	Finderが利用するフォルダ情報ファイル (MacOSのみ)
.ssh (ディレクトリ)	ssh接続のための鍵ファイルなどを保管する。秘密鍵を含むことがあるため取扱いに注意が必要
.svn (ディレクトリ)	Subversionの作業コピー管理ディレクトリ
.git (ディレクトリ)	Gitのリポジトリ管理ディレクトリ

このスクリプトではドットファイルのリストをlsコマンドで作りますが、そのための事前準備として、❶でIFS変数を変更しています。❶は文の途中で改行してゴミが入っているかのように見えますが、正しい書き方です。これは、はじめて見る人にはとても奇妙な書き方に思われることでしょう。

IFSとはInternal Field Separatorの略で、シェルが区切り文字として解釈する文字を設定する特殊な変数です。デフォルトでは改行・タブ・スペースが設定されています。❶は、このIFSに改行のみを代入し、以降のスクリプトではシェルの区切り文字として改行だけを用いて、スペースとタブを区切り文字として扱わないようにするための処理です。

このサンプル例ではファイル名を扱うため、ファイル名にスペースを含んでいる場合にはひとまとめの文字列として扱い、スペースを区切りと解釈したくありません。そのためIFSとして、このように改行だけを設定してスペースを含めていないのです。変数代入が❶のように2行に分かれるというのは、他の言語ではなかなか見られない書き方ですが、シェルスクリプトのIFS設定の場面では比較的よく使われる記法です。

さて、ドットファイルはlsコマンドを使えば簡単にリストが得られると思われるでしょうが、これが意外にややこしく面倒です。単純に* (アスタリスク) を使って、「ls *」のようにワイルドカード指定をしても、ドットファイルは対象となりません。これはシェルの仕様として、ドットファイルは*にマッチしないようになっているためです。

lsコマンドのワイルドカード指定でドットファイルを得るには、.*と「ドット＋ワイルドカード」を明示的に指定する方法があります。しかしこの場合は、カレントディレクトリを意味する. (ドット) と、親ディレクトリを意味する.. (ドットドット) も対象となってしまいます。

そこでこのサンプル例では❷にて、lsコマンドの**-Aオプション**でドットファイルを含めてカレントディレクトリ配下のファイルすべてを表示し、それをcase文でより分ける、という手法でドットファイルだけを抽出することにしました。ls -Aならば、カレントディレクトリや親ディレクトリは表示されないため、先頭文字がドットかそうでないかの単純な判断だけでドットファイルを抽出できます。

なお❷ではこの際、ファイルなのかディレクトリなのかを区別するためにlsコマンドの**-Fオプション**も一緒に利用しています。-Fオプションを付けるとディレクトリの最後には/(スラッシュ)が付くため、これを見てディレクトリなのかファイルなのかを判断できます。

実際にファイルかディレクトリかを判別するのは、❸の**case文**で行っています。シェルスクリプトで文字列をマッチさせて処理分岐させたいときには、このようにcase文を使うと手軽に書けます。ファイル名がドットで始まってスラッシュで終わっている場合(.*/)にはディレクトリと解釈して"dot directory"と表示し、それ以外でドットから始まっている場合(.*)にはファイルとみなして"dot file"と表示しています。

注意事項

・このサンプル例でさらに続けて何らかの処理を行う場合、IFSを改行のみとしていることが悪影響を及ぼすことがあります。そのためIFSを一時的に変更する際は、最初に現在のIFSを保存しておいて、後で値を元に戻す、という手法が一般的によく使われます。

```
IFS_BACKUP=$IFS
IFS='
'

# (ここで必要な処理を行う)

IFS=$IFS_BACKUP
```

・IFSに改行を設定したい場合、次のように書くこともできます。このほうが改行で分かれた書き方をせずに済むため、見やすいかもしれません。ただし、POSIX準拠の書き方ではないため、Ubuntuのshなど一部の環境では動作しない場合もあります。

```
IFS=$'¥n'
```

No. 049 二重起動が可能な一時ファイルを作成する

利用コマンド
date, cat

キーワード
テンポラリファイル, 一時ファイル, プロセスID

いつ使うか
スクリプトを同時実行させるとき、一時ファイルを重複しないようにしたいとき

実行例

```
$ ./tmppid.sh
Sat Dec 14 22:50:16 JST 2013
```

スクリプト

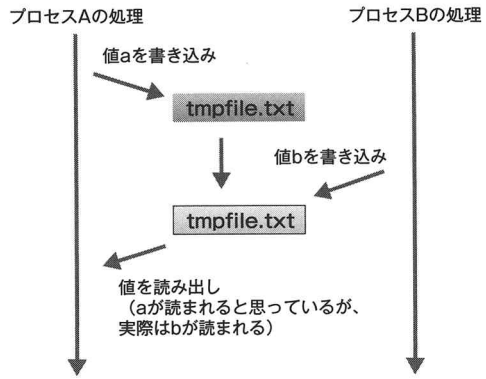
```
#!/bin/sh

tmpfile="tmp.$$" ―――❶

date > $tmpfile ―――❷
sleep 10

cat $tmpfile ―――❸
rm -f $tmpfile
```

解説

このスクリプトは、プログラム内で一時的に利用するテンポラリファイル(以下一時ファイル)を扱うものです。スクリプト起動時にdateコマンドで現在の日付を一時ファイルに書き出し、10秒待ってからその値をファイルから読み込んで出力します。このサンプルでは、シェル変数tmpfileで指定している一時ファイル名を、**プロセスID**($$)を用いて生成していることがポイントです。

シェルスクリプトでは、途中の計算結果や処理結果などを後で利用するために、一時ファイルを使う場面が多々あります。この際に気を付けなければいけないのが、二重起動された場合の処理です。例えば、もしtmpfile.txtという名前の固定ファイル名で一時ファイルを作っていると、次のように後から起動したプロセスで値が上書きされてしまうことがあります。

●一時ファイルが重複しているとマズいことになる



これを防ぐには、同じシェルスクリプトを実行しても、それぞれのプロセスが違う名前で一時ファイルを作るようにしておくことが必要です。その際によく使われる手法が、本サンプルで用いている、**拡張子を.$$とする**やり方です。

$$とはシェルの特殊変数で、実行しているシェルのプロセスIDを値として持っています。プロセスIDとはOS上でプロセスごとに振られる一意の数値で、同じシェルスクリプトであってもプロセスが違えば違う値となります。❶ではこの一意性を利用して、一時ファイルのファイル名としてプロセスIDを利用しています。続いて❷でこの一時ファイルに**dateコマンド**の結果を出力して、現在時刻を保存しています。

このサンプル例は途中に**sleepコマンド**を挟むぶん時間がかかるスクリプトであるため、他の人が同時に実行して何重にも起動されるかもしれません。しかしそれらはすべて違うプロセスになりますから、それぞれ違うプロセスIDを持ち、結果としてtmp.$$で指定する一時ファイル名もすべて違うファイル名になります。そのため、❸で一時ファイルの中身を出力する際も、それら多重起動しているプロセスがお互いに影響を及ぼすことのないように扱うことができるのです。

このように、プロセスごとに違う値にしたい場合は、この$$を使うのがセオリーですから覚えておいたほうがよいでしょう。

注意事項

・ランダムかつ一意なファイル名を持つ一時ファイルを作るコマンドとして、mktempというコマンドもあります。このサンプルのようにプロセスIDを用いた場合、悪意のある攻撃者が比較的容易に一時ファイル名を推察できてしまうため、Webアプリケーションなどではセキュリティを考慮してmktempコマンドが推奨される場合もあります。

No. 050 sedでファイル置換を行う際、シンボリックリンクを実ファイルで置き換えないようにする

利用コマンド
sed, readlink

キーワード
シンボリックリンク, ファイル置換, 実体ファイル

いつ使うか
sedコマンドの-iオプションで、シンボリックリンクを対象とするとき

実行例

```
$ ls -F
orig/  sed-symlink.sh*   target.txt@ ―― カレントディレクトリにシンボリックリンクtarget.txt@があり、orig/ディレクトリの中に実体ファイルがある
$ ls orig/
target.txt
$ ./sed-symlink.sh
$ ls -F
orig/  sed-symlink.sh*   target.txt@
$ ls -F orig/
target.txt       target.txt.bak ―― 実体ファイルが書き換えられた
```

スクリプト

```
#!/bin/sh

filename="target.txt"

if [ ! -e "$filename" ]; then ―― ❶
  # 対象ファイルが存在しなければエラー終了
  echo "ERROR: File not exists." >&2
  exit 1
elif [ -h "$filename" ]; then ―― ❷
  # 対象ファイルがシンボリックリンクならば、readlinkコマンドで
  # 実体ファイルに対して処理を行う
  sed -i.bak "s/Hello/Hi/g" "$(readlink "$filename")" ―― ❸
else
  sed -i.bak "s/Hello/Hi/g" "$filename"
fi
```

解説

このスクリプトは、**sedコマンド**の**-iオプション**でファイルの上書き置換を行う際、シンボリックリンクをそのままリンクとして扱うものです。この環境では、カレントディレ

クトリにtarget.txtという名前のシンボリックリンクがあり、このリンクはorigディレクトリ内の実体ファイルに向いていると仮定しています。これはlsコマンドでも-lオプションを付けると確認できます。次のような表示を見たことがあるかもしれません。

シンボリックリンクは実ファイルを指す

```
lrwxrwxrwx.  1 user1.user1  236 Dec 21 17:02 target.txt  ->  orig/target.
txt
```

サンプルでは、target.txtというシンボリックリンクを対象としてsedコマンドを実行し、ファイル内の"Hello"を"Hi"に置換しています。

さて、sedコマンドはフィルタコマンドであり、通常は入力元のファイルを上書きしません。そのため置換後の結果で元ファイルを上書きするには、次のように中間ファイル(ここではtmp)を作る必要があります。

通常は中間ファイルを作ってから上書きする

```
$ sed "s/Hello/Hi/g" target.txt > tmp
$ mv tmp target.txt
```

しかしこの中間ファイルを作りたくないケースのために、sedコマンドには元ファイルを上書きする-iオプションが用意されています。

なお-iオプションを使う際には、上書きという破壊的処理に備えてバックアップファイルの指定を行います。例えば、元ファイルに拡張子.bakを付けてバックアップをとり、上書きする場合は次のようにします。これにより、置換後の結果で上書きしたtarget.txtと、元ファイルのバックアップファイルtarget.txt.bak、の2つのファイルができます。

```
sed -i.bak "s/Hello/Hi/g" target.txt
```

しかし-iオプションは、シンボリックリンクを対象とするとややこしいことになります。-iオプションは対象がシンボリックリンクであっても上書きをしてしまうため、このサンプル例では、カレントディレクトリにあるシンボリックリンクを置換後の内容で上書きしてしまいます。結果として、次の2つのファイルができてしまいます。これは管理上、問題となるでしょう。

- カレントディレクトリの、置換後のtarget.txt
- orig/ディレクトリ内の、元のファイルtarget.txt

そこでこのサンプル例では、対象がシンボリックリンクかどうかを判定して処理を変えています。まず❶では、ファイルが存在するかを**testコマンド**の**-e演算子**(→P.110参照)で判定し、ファイルがなければ**exitコマンド**で終了しています。

続いて❷で、testコマンドの**-h演算子**で対象ファイルがシンボリックリンクかどうかをチェックしています。もしシンボリックリンクだった場合、❸で**readlinkコマンド**を用いて実体ファイルのパスを得て、この実体ファイルに対してsedコマンドを実行します。なお、readlinkコマンドとは次のように、シンボリックリンクを引数にとり、実体ファイルへのパスを表示するコマンドです。

readlinkコマンドでシンボリックリンクを調べる

```
$ readlink target.txt
original/target.txt
```

このように表示された場合、カレントディレクトリのtarget.txtはorig/target.txtへのシンボリックリンクです。

こうしてreadlinkコマンドの出力結果をsedコマンドで処理すれば、元の実体ファイルを上書きしますからシンボリックリンクもそのまま残り、ファイルを正しく扱えるわけです。

注意事項

- Linuxのsedには、シンボリックリンクのリンク先を追ってくれる**--follow-symlinks**というオプションがあります。これを使えば、サンプル例は次のように簡単に書くことができます。この場合、対象ファイルが通常ファイルであればそのファイルを処理しますし、シンボリックリンクならばリンク先のファイルを処理するため、リンクは壊れません。

```
sed -i.bak --follow-symlinks "s/Hello/Hi/g" "$filename"
```

関連項目

042　処理開始前に、実行権限をチェックして正常動作できることを確認してから実行する

CHAPTER

04

日付処理

ログファイルなどのファイル名には日付が使われることが多いため、シェルスクリプトではこれらファイルの操作のために「今日が月末かどうかを判定する」「現在の前月を取得する」など、日付処理が必要とされる機会が多くあります。この章では、dateコマンドを用いて日付処理をおこなうサンプル例を紹介します。LinuxとMac/FreeBSDではdateコマンドの書式に違いがあるため、この差異も合わせて解説します。

No. 051 dateコマンドで日付の比較と取得を行う

利用コマンド
date, expr

キーワード
日時, UNIX時間, エポック秒

いつ使うか
日時表記のテキストが2つあり、数値計算して何日の差があるのかを計算したいとき

実行例

```
$ ./date-epoch.sh
day1(2012/04/01 10:49:41): 1333244981
day2(2012/03/30 08:31:52): 1333063912
day interval:
2
```

スクリプト

```
#!/bin/sh

# 比較したい2つの日時を変数として定義
day1="2012/04/01 10:49:41"
day2="2012/03/30 08:31:52"

# 日付からepoch秒を取得するには+%sを利用する(Linux)
# ※ -dオプションはFreeBSD/Macでは使えない
day1_epoch=$(date -d "$day1" '+%s')
day2_epoch=$(date -d "$day2" '+%s')

echo "day1($day1): $day1_epoch"
echo "day2($day2): $day2_epoch"

# 2つの日付のepoch秒を引き算して、
# 1日=24時間=1440分=86400秒で割れば何日違うかを計算できる
echo "day interval: "
day_interval=$(expr ¥( $day1_epoch - $day2_epoch ¥) / 86400)
echo $day_interval
```

❶ ❷ ❸ ❹

解説

このスクリプトは、文字列として与えられた2つの日付の差をUNIX時間に変換して比較し、その差が何日あるかを計算してシェル変数day_intervalに代入しています。利用ケースとしては、例えば申し込み日付から5日以上経過しているユーザには何らかのメール

を送るなどのプログラムで、このday_intervalが5以上ならばメール送信スクリプトを実行する、などの使い方が考えられます。

さて、UNIXサーバを運用していると、以下のように現在日時を処理しなければならない場面は非常に多くあります。

- ログファイルのファイル名から、古い日付を持つファイル名の判定
- 月次処理の日付判定
- 受け付け日時／解除日時の登録など
- ファイルのタイムスタンプ判断

こんなとき、UNIXでは一般的に「UNIX時間」という値で日時を処理するケースが多くあります。これは、**UNIXエポック**(epoch)と呼ばれる1970年1月1日0時0分0秒からの経過秒数を表すもので、エポック秒とも呼ばれます。例えば2000年1月1日0時0分0秒は、UNIX時間で946652400です。

一般的に日付をコンピュータで扱う際、年月日などを書く形式が国や文化によって違うという問題点があります。例えば日本では「2013年12月15日」と書くところ、英語では「Dec 15 2013」と書いたりします。このように言語だけでなく表記順も異なります。これでは扱う環境によって日付処理を書きなおさなくてはならずとても不便です。しかしUNIX時間を使えば、どんな国や文化でも表記は同一ですし、値も単なる整数値で扱いやすいため、UNIX環境では広く用いられている時刻フォーマットです。

このサンプル例では、まず❶で比較すべき2つの日付を変数に代入し、❷で**dateコマンド**の出力形式を'+%s'と指定することにより、それらの日付のUNIX時間を算出しています。この際、日時の指定に**-dオプション**を利用していますが、これはLinuxのみで使える表記です。FreeBSDやMacをお使いの場合は、「注意事項」を参照してください。

❸で、テキスト表記の日時とUNIX時間の表記を、echoコマンドで出力しています。UNIX時間はPerlやRubyなどのスクリプト言語でもよく使われますから、そのような外部スクリプトと連携したい場合は、このUNIX時間を渡すとよいでしょう。

❹では、day1とday2で何日の差があるかを計算しています。UNIX時間は単なる秒数なので、単純に**exprコマンド**で引き算をすれば時刻の差が計算できます。その差を、1日=24時間=1440分=86400秒で割れば、何日の差があるのかを計算できます。

注意事項

- このサンプル例では、day2のほうが未来の日付の場合は、日付差としてマイナス表示をします。そのためもし日付差の絶対値がほしい場合は、どちらが未来か(どちらのUNIX時間が大きいか)を判断して、大きいほうから小さいほうを引く必要があります。

```
day1(2012/03/30 08:31:52): 1333063912
day2(2012/04/01 10:49:41): 1333244981
day interval:
-2
```

・FreeBSDおよびMacなどのBSD系UNIXのdateコマンドと、Linuxのdateコマンドは、オプションに多くの差があります。スクリプトの❷で用いている-dオプションは、FreeBSDおよびMacでは使えません。そのためFreeBSDおよびMacの場合は、日付設定ではなく日付表示を行う-jオプションと、フォーマット指定をする-fオプションを用いて、次のように❷の部分を修正してください。

```
# 日付から epoch 秒を取得するには +%s を利用する (BSD/Mac)
day1_epoch=$(date -j -f "%Y/%m/%d %H:%M:%S" "$day1" '+%s')
day2_epoch=$(date -j -f "%Y/%m/%d %H:%M:%S" "$day2" '+%s')
```

ここで用いている%Yや%mなどの表記は、P.89のdateコマンドの説明を参照してください。

・UNIX時間は扱いが便利なため、日付時刻を扱うデータベースでもよく利用されます。例えばMySQLでは、UNIX時間から日付文字列へと変換する**FROM_UNIXTIME()**という関数が標準で用意されています。

MySQLでUNIX時間を扱う

```
mysql> SELECT FROM_UNIXTIME(1333063912);
+---------------------------+
| FROM_UNIXTIME(1333063912) |
+---------------------------+
| 2012-03-30 08:31:52       |
+---------------------------+
```

関連項目

030 あるディレクトリ内の、n日前からm日前までに更新されたファイル一覧を取得する
033 ファイルをバックアップする際にファイル名に日時を入れる

No. 052 今日が月末日かどうかを判定する

利用コマンド
date

キーワード
末日, 月末

いつ使うか
月末の〆処理などのため、今日が月末日かどうかをスクリプト内で判定したいとき

実行例

```
$ ./monthday.sh ―――― 月次レポート作成の外部スクリプトを実行
```

スクリプト

```
#!/bin/sh

tomorrow=$(date "+%d" -d '1 day') ――――❶

if [ "$tomorrow" = "01" ]; then ――――❷
  # 今日が月末ならば、月次レポートを作成するための
  # 外部スクリプトを実行
  ./monthly_report.sh
fi
```

解説

このスクリプトは、現在の日付を判断し、月末ならば月次レポートを集計するmonthly_report.shという外部スクリプトを実行するものです。集計処理や〆処理などで、月末日だけに実行するプログラムやバッチ処理は多いでしょう。しかし定期実行するためのcronの仕組みでは、月初は「1日」に動くように設定すれば済むのですが、月末日を指定する仕組みがありません。このためよく使われる手法が、次のようなやり方です。

1) cronでは、スクリプトが毎日起動するように設定しておく
2) スクリプト内で、今日が月末かを判断するロジックを入れておき、月末の場合のみ処理を行う

そのため、「今日が末日かどうか」を判定するという処理は何かとよく使われます。そこでここでは、月末日の判断ロジックを紹介し、月末ならば月次レポートを作成する外部スクリプトを実行する、という例を見てみます。

月末の日は、例えば1月なら31日であり、2月なら28日(うるう年ならば29日)ですから、月と年によって変わるため厳密に判定しようとすると意外に大変です。そこでこのサンプ

ル例ではもっと単純なロジックとして、「翌日が1日ならば、今日は末日」を採用しています。

❶で、**dateコマンド**を用いてシェル変数tomorrowに翌日を取得しています。ここでは"+%d"として、dateコマンドの出力形式を日だけにするよう指定しています。この%dなどの表記は、P.89のdateコマンドの説明を参照してください。

また❶では、あわせて**-dオプション**に'1 day'として翌日を指定しています。dateコマンドでは、-dオプションをこのように利用して現在時刻の前後の日付を指定することができます。例えば3日前の日がほしいならば、次のように'3 days ago'とします。

```
date "+%d" -d '3 days ago'
```

なお、-dオプションはLinuxのみで使える表記です。FreeBSDやMacをお使いの場合は、「注意事項」を参照してください。

❷で、シェル変数tomorrowに代入されている明日の日付が、文字列"01"と等しいかどうかをif文で判定しています。もし得られた日付が"01"ならば、それは今日が月末日だということです。そのためif文の中で、末日のみ行う〆処理などを実行することができます。サンプル例では、月次レポートを作成するスクリプトmonthly_report.shがカレントディレクトリにあると仮定し、これを実行しています。

注意事項

- FreeBSDおよびMacなどのBSD系UNIXのdateコマンドと、Linuxのdateコマンドは、オプションに多くの差があります。スクリプトの❶で、翌日を指定している-d '1 day'という表記は、FreeBSDおよびMacでは使えません。そのためFreeBSDおよびMacの場合は、時間をずらす-vオプションを用いて記述してください。

```
tomorrow=$(date -v+1d "+%d")
```

-v+1dは、「-vオプションに+1dを指定」を意味します。ここでは+1dだけずらすことで「1日後」を指定しています。-vオプションの詳しい使い方は、dateコマンドのマニュアルを参照してください。

関連項目

033 ファイルをバックアップする際にファイル名に日時を入れる
051 dateコマンドで日付の比較と取得を行う

No. 053 現在の前月を取得して、前月に作成されたログファイルを一括アーカイブする

利用コマンド
date, tar

キーワード
前月, 末日

いつ使うか
現在の日付から、先月の年月表記(YYYYMM)を組み立てて、日付をファイル名に含むログファイルをまとめてアーカイブしたいとき

実行例

```
$ ./lastmonth.sh
/var/log/myapp/access.log-20130201
/var/log/myapp/access.log-20130218
/var/log/myapp/access.log-20130222
/var/log/myapp/access.log-20130228
```

スクリプト

```
#!/bin/sh

logdir="/var/log/myapp"

# 今月の15日の日付を取得する
thismonth=$(date '+%Y/%m/15 00:00:00') ―――――――――――――――――――――❶

# 先月の日付をYYYYMMで取得する。
# 1 month agoは先月の同日をとるため、末日とならないよう
# 変数thismonthに15日を指定した
last_YYYYMM=$(date -d "$thismonth - 1 month ago" '+%Y%m') ―――――――――――❷

# 先月のログファイルをまとめてアーカイブ
tar cvf ${last_YYYYMM}.tar ${logdir}/access.log-${last_YYYYMM}* ―――――――❸
```

解説

このスクリプトは、現在の日付から先月の年月表記(YYYYMM)を組み立て、日付がファイル名に付いているログファイルから、先月ぶんを抽出してアーカイブするものです。

ここでは、現在が2013年3月30日であるとします。また/var/log/myappというディレクトリ内には次のように、ファイル名に日付を持ったログファイルが「access.log-年月日」の形で存在するものと仮定しています。

ログファイルを確認

```
$ ls /var/log/myapp
access.log-20130130  access.log-20130227
access.log-20130131  access.log-20130228
access.log-20130201  access.log-20130301
(省略)
```

このようにログファイルにファイル名として年月日が付いている場合に、月初の月次バッチ処理として、先月のファイルをまとめてアーカイブするため、先月の年月表記(YYYYMM)を取得したいというケースがよくあります。これを取得してみましょう。

先月の日付表記を得たい場合、Linuxでは**dateコマンド**の**-dオプション**で、1ヶ月前すなわち"1 month ago"を指定するのが便利です。しかし、この"1 month ago"は、注意して使わないと思わぬ結果を招きます。

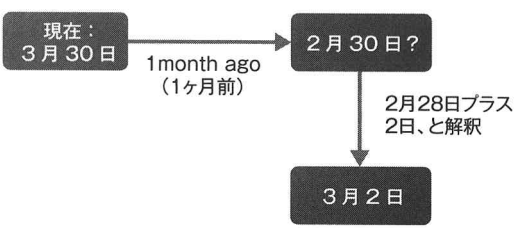
dateコマンドの"1 month ago"でよくある失敗

```
$ date "+%Y/%m/%d %H:%M:%S"
2013/03/30 00:00:05 ──────────────────────────── 現在の日付
$ date "+%Y/%m/%d %H:%M:%S" -d "1 month ago"
2013/03/02 00:00:06 ──────────────────────────── 2月28日にならない
```

上記の例のように3月の月末である3月30日に「date -d "1 month ago"」を実行しても、2月の月末である2月28日は返りません。これは、Linuxのdateコマンドで"1 month ago"を指定すると、「前月の同じ日」を取得するためです。この例では現在が3月30日のため「2月30日」を取得しようとしますが、そんな日は存在しないため、2月28日から数えて2日後、つまり3月2日を取得してしまうのです。

結果が3月2日となる仕組み

Linuxのdateコマンドの、1 month ago（1ヵ月前）



これを防ぐためには、どの月にも存在する日を指定して"1 month ago"指定をすればよいことになります。つまり1日～28日のどれかを指定します。このあいだの日ならどこでもよいのですが、サンプル例では中間あたりということで15日を選択しました❶。

なお、このような月末日の考え方は、Linuxのdateコマンドの特徴です。FreeBSDおよびMacのdateコマンドでは動きが異なるため、本項の「注意事項」を参照してください。

❷で、先ほど15日を指定した今月の日付から、先月を得るためにdateコマンドで"1 month ago"の日付を取得しています。'+%Y%m'を指定することで、先月の年月表記(YYYYMM)が得られます。

❸で、先月日付を元に、ログファイルをアーカイブしています。これは現在が3月30日ならば、実際に変数展開されると次のようになり、先月である2013年2月のログファイルをアーカイブできるわけです。

```
tar cvf 201302.tar /var/log/myapp/access.log-201302*
```

注意事項

- FreeBSDおよびMacなどのBSD系UNIXのdateコマンドと、Linuxのdateコマンドは、オプションに多くの差があります。FreeBSDおよびMacで先月を取得したい際、スクリプトの❷で先月を指定している-d '1 month ago'という表記は使えません。そのためFreeBSDおよびMacの場合は、時間をずらす-vオプションを用いて次のように記述してください。

```
last_YYYYMM=$(date -v-1m "+%Y%m")
```

-v-1mは、「-vオプションに-1mを指定」を意味します。ここでは-1mずらすことで「先月」を指定しています。-vオプションの詳しい使い方は、dateコマンドのマニュアルを参照してください。

- FreeBSDおよびMacの-v-1mで指定する「1ヶ月前」は、Linuxの"1 month ago"指定と違って、月末日はそのまま先月の末日となります。そのため15日にずらすテクニックは必要ありません。

関連項目

033 ファイルをバックアップする際にファイル名に日時を入れる
052 今日が月末日かどうかを判定する

No.
054 今年がうるう年かどうかを調べる

利用コマンド
expr, test, ls

キーワード
うるう年, 剰余

いつ使うか
日付をファイル名に持つログファイルから、2月末日に作られたものを選択したいとき

実行例

```
$ ./leapyear.sh
leap year:2012
/var/log/myapp/access.log-20120229
```

スクリプト

```
#!/bin/sh

# 現在の西暦を4桁で取得
year=$(date '+%Y')                                    ❶

logfile="/var/log/myapp/access.log-"

# 西暦を割った余り(剰余)を計算する
mod1=$(expr $year % 4)   ┐
mod2=$(expr $year % 100) ├──── ❷
mod3=$(expr $year % 400) ┘

# うるう年かどうかの判定
if [ $mod1 -eq 0 -a $mod2 -ne 0 -o $mod3 -eq 0 ]; then ── ❸
  echo "leap year:$year"
  ls "${logfile}${year}0229"                           ❹
else
  echo "not leap year:$year"
  ls "${logfile}${year}0228"                           ❺
fi
```

解説

このスクリプトは、2月末日のファイルを表示するために、現在がうるう年かそうでないかを判別するものです。ここでは、"access.log-20120315"のように、ファイル名に年月日が入ったログファイルが/var/log/myappというディレクトリに蓄積されていると仮定しています。

月次〆処理の結果を確認するなど、月末のログファイルを参照する機会は多いでしょう。そしてこのサンプル例のように、ファイル名に日付が入っているログファイルなどを扱う際、2月末日をどうやって取得するかはプログラマを悩ませる世界共通の問題です。

ここでまず、うるう年を判定する条件を確認しておきましょう。以下の3つとなります。

(1) 西暦年が4で割り切れる年は、うるう年。
(2) ただし、西暦年が100で割り切れる年は、うるう年ではない。
(3) ただし、西暦年が400で割り切れる年は、うるう年。

特に3番目の条件はあまり知られておらず、2000年に多くのバグを発生させる原因となりました。2000年は、この3番目の条件に引っかかる、400年に一度の珍しいうるう年だったのです。

このサンプル例では、まず❶で現在の西暦年を4桁で取得してシェル変数yearに代入しています。この**dateコマンド**での+%Yなどの使い方は、P.89を参照してください。

❷で、西暦年を割った余りを**exprコマンド**で計算しています。exprコマンドで**%演算子**を使うことで、割り算の余り(剰余)を計算できます。ここでは条件1、2、3としてそれぞれの余り(剰余)をシェル変数mod1、mod2、mod3に代入しています。

❸で、先にあげたうるう年の条件をif文で判定しています。「割り切れる」ということはすなわち「余り(剰余)が0」ということですから、ここでは0と変数の値とを比べています。

値が等しいかどうかは**-eq演算子**で、等しくないかどうかは**-ne演算子**で判断できます。また**-a**はAND(かつ)、**-o**はOR(または)を示す演算子です。つまり❸の条件式全体を日本語で書くと、『「mod1の値が0」かつ「mod2の値は0ではない」』または『mod3の値が0である』、という条件を意味しています。なお、testコマンドの数値判定を行う演算子については次にあげておきましたので参考にしてください。

testコマンドで比較を行う演算子

記述方法	意味
変数1 -eq 変数2	変数1と変数2が等しければ真
変数1 -ne 変数2	変数1と変数2が等しくなければ真
変数1 -lt 変数2	変数1が変数2未満ならば真
変数1 -le 変数2	変数1が変数2以下ならば真
変数1 -gt 変数2	変数1が変数2より大きければ真
変数1 -ge 変数2	変数1が変数2以上ならば真

ちなみにltは英語の"less than"、leは"less than or equal to"、gtは"greater than"、geは"greater than or equal to"を指しています。

これでうるう年かどうか判定できたので、2月末日のファイルを選択することができます。❹と❺で、うるう年の場合には2月29日のログファイルを、うるう年でない場合は2月28日のログファイルをlsコマンドで表示しています。

注意事項

・本項で利用したtestコマンドの演算子-a(AND)と-o(OR)の優先度について、補足しておきましょう。他のプログラミング言語と同様にシェルスクリプトのtestコマンドも、ANDのほうがORよりも優先度が高い演算子となっています。もしORの方の優先度をANDよりも高くしたい場合には、通常の数式のようにカッコでくくれば優先して評価されます。ただしこの際、シェルにカッコが解釈されないように\でエスケープして、\(··· \)のようにカッコを書く必要があります。

```
# 先の-a(AND)よりも後ろの-o(OR)が優先される書き方
if [ $a -eq 0 -a \( $b -ne 0 -o $c -eq 0 \) ]; then
```

上記の例は、「$aが0、かつ、$bが0ではないまたは$cが0」のときに真となります。
なおtestコマンドはシェルビルトインと外部コマンドの2種類がありますが、シェルスクリプトではシェルビルトインのtestコマンドが利用されます。そのためマニュアルを読む際は、man shとしてシェルのマニュアルを見れば、testコマンドの解説が書かれています。

関連項目

033 ファイルをバックアップする際にファイル名に日時を入れる

CHAPTER

05

ネットワーク

現在のUNIXシステムの構成要素として、ネットワークは必須のものとなっています。この章では、pingコマンドやncコマンドなどを利用して、シェルスクリプトでネットワークを扱う様々な事例を紹介します。本章のサンプルを用いて、ネットワークのテストや速度測定、DNSでの名前解決、ファイル転送の自動化などをおこなうことができます。各コマンドの使い方も、あわせて紹介しています。

No. 055 デフォルトゲートウェイにpingが通るかテストする (Linux)

利用コマンド
route, awk, ping

キーワード
ICMP, デフォルトゲートウェイ

いつ使うか
デフォルトゲートウェイを自動で取得して、pingコマンドでネットワーク疎通を確認したいとき

実行例

```
$ ./gwping-linux.sh
[Success] ping -> 192.168.1.1
```

スクリプト

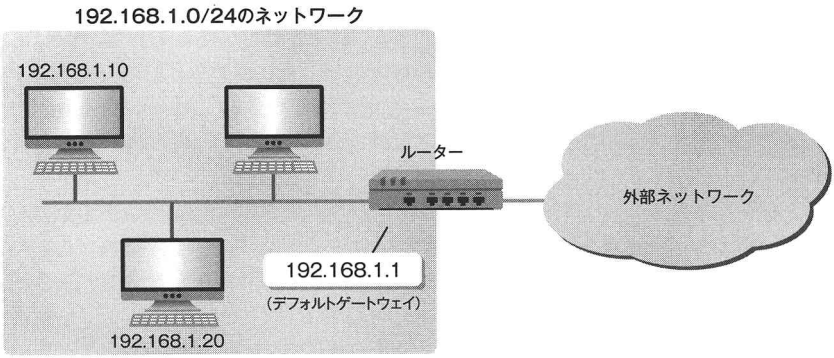
```
#!/bin/sh

# routeコマンドの出力からデフォルトゲートウェイを取得。
# カラム1が"0.0.0.0"の行の、カラム2を取り出す
gateway=$(route -n | awk '$1 == "0.0.0.0" {print $2}') ―― ❶

# デフォルトゲートウェイにping実行
ping -c 1 $gateway > /dev/null 2>&1 ―― ❷

# pingコマンドの終了ステータスで成功・失敗判断
if [ $? -eq 0 ]; then ―― ❸
  echo "[Success] ping -> $gateway"
else
  echo "[Failed] ping -> $gateway"
fi
```

解説

このスクリプトは、Linuxサーバからデフォルトゲートウェイへのネットワーク疎通をテストするものです。**pingコマンド**で**ICMPパケット**を送信し、ネットワークが正常に通信できるかどうかを確認します。

まず、このサンプル例のテーマである**デフォルトゲートウェイ**について簡単に解説しておきましょう。デフォルトゲートウェイとは、ネットワークのアクセス経路において、外部ネットワークとの出入り口になる機器のことです。これは一般的にはルータになります。

デフォルトゲートウェイとは



例えば上記の図で、「192.168.1.10」から「192.168.1.20」宛てに通信する際には、同一セグメント内であるため、デフォルトゲートウェイは利用されません。

一方、自分が所属しないネットワーク、例えば「192.168.2.10」宛てに通信する場合には、マシンはデフォルトゲートウェイである「192.168.1.1」（ここではルータ）宛てにパケットを送信します。ルータは、「192.168.2.10」という宛先IPアドレスに従って、適切なルーティングを行います。

サーバ管理においてネットワークに何か異常がある際に、まずはデフォルトゲートウェイにpingを打って疎通を確認するというのは大変よく行われる作業です。

Linuxサーバでデフォルトゲートウェイを調べるにはいくつか方法がありますが、ここではルーティングテーブルを表示する**routeコマンド**の出力を利用してみます。なおrouteコマンドの出力はLinuxとFreeBSD/Macで異なるため、ここではLinuxでの例を紹介します。FreeBSD/Macをご利用の場合は、次項を参照してください。

routeコマンドは、次のように**-nオプション**だけを付けると、現在の経路テーブルの内容をホスト名ではなくIPアドレスで表示します。

Linuxのrouteコマンド出力例

```
$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 eth0
169.254.0.0     0.0.0.0         255.255.0.0     U     1002   0        0 eth0
224.0.0.0       0.0.0.0         240.0.0.0       U     0      0        0 eth0
0.0.0.0         192.168.1.1     0.0.0.0         UG    0      0        0 eth0
```

上記には多くの項目がありますが、ここではDestinationとGatewayの値を利用します。Destinationは宛先ネットワークを指し、このサンプル例ではデフォルトゲートウェイを取得したいため"0.0.0.0"の行（最終行）を読みます。この行のGatewayが192.168.1.1にな

っていますから、192.168.1.1がこのマシンのデフォルトゲートウェイです。

❶にて、routeコマンドの出力からテキスト加工してIPアドレスを取り出すために**awkコマンド**を利用しています。awkでは、$1、$2…という変数がそれぞれ第1カラム、第2カラムを指します。ここでは第1カラムDestinationが"0.0.0.0"である行のみを出力するようにフィルタして、アクションを示す中カッコ{}内で第2カラムGatewayを**printfコマンド**で表示しています。この出力を、コマンド置換記法**$()**でシェル変数gatewayに代入しています。

❷で、取得したデフォルトゲートウェイのIPアドレス宛てにpingコマンドを実行しています。Windowsのpingコマンドは4回ICMPパケットを送ると自動で終了しますが、Linuxのpingコマンドは、オプションを指定しないとICMPパケットを永遠に送り続けてしまいます。そのためここでは**-cオプション**を利用して1回だけ実行するように指定しています。また、このサンプル例ではpingコマンドの終了ステータスのみ利用し、出力は必要ないため、標準出力および標準エラー出力を**/dev/null**へとリダイレクトして捨てています。

❸で、pingコマンドの実行結果を判断しています。pingコマンドで送信したICMP Echo Requestに対して、正常にICMP Echo Replyが返ってきていれば、pingコマンドの終了ステータスは0となっています。シェルの**特殊変数$?**にはコマンドの終了ステータスが入っていますので、この値が0かどうかで成功したか失敗したかを判断してメッセージを出し分けています。

注意事項

- Linuxサーバ自体にデフォルトゲートウェイが設定されていない場合、❷のpingコマンドはシェル変数gatewayが空であることから、引数エラーとなります。そのためこのサンプル例で「Failed」となった場合は、以下のような可能性があります。
  - ▶サーバの設定自体に問題がある（ネットワークの設定が誤っているなど）
  - ▶途中のネットワーク経路に問題がある（LANケーブルが抜けているなど）
  - ▶デフォルトゲートウェイの機器に問題がある（電源が落ちているなど）
- ネットワークの疎通を確認するならば、pingは1回ではなく複数回実行して結果を見たほうがよいでしょう。pingを複数回実行するサンプルはP.323を参照してください。

関連項目

058 arpテーブルから指定IPアドレスに対応するMACアドレスを表示する
114 サーバのping監視を行う

No. 056 デフォルトゲートウェイにpingが通るかテストする(FreeBSD/Mac)

利用コマンド
netstat, awk, ping

キーワード
ping, ICMP, デフォルトゲートウェイ

いつ使うか
デフォルトゲートウェイを自動で取得して、pingコマンドでネットワーク疎通を確認したいとき

実行例

```
$ ./gwping-bsd.sh
[Success] ping -> 192.168.1.1
```

スクリプト

```
#!/bin/sh

# netstatコマンドの出力からデフォルトゲートウェイを取得。
# カラム1が"default"の行の、カラム2を取り出す
gateway=$(netstat -nr | awk '$1 == "default" {print $2}') ―― ❶

# デフォルトゲートウェイにping実行
ping -c 1 $gateway > /dev/null 2>&1 ―― ❷

# pingコマンドの終了ステータスで成功・失敗判断
if [ $? -eq 0 ]; then ―― ❸
  echo "[Success] ping -> $gateway"
else
  echo "[Failed] ping -> $gateway"
fi
```

解説

このスクリプトは、FreeBSDやMacマシンからデフォルトゲートウェイへのネットワーク疎通をテストするものです。**pingコマンド**でICMPパケットを送信し、ネットワークが正常に通信できるかを確認します。pingコマンドによるデフォルトゲートウェイとの通信テストについては、サンプル055で説明していますのでそちらも参照してください。

Linuxでは、ルーティングテーブルを表示するためには**routeコマンド**を引数なしで実行しますが、FreeBSDで同様のコマンドを実行してもエラーとなります。

これはLinuxとBSD系で、routeコマンドの仕様が大きく異なるためです。FreeBSDやMacでは、routeコマンドは基本的に経路制御を行うコマンドで、単にルーティングテー

ブルを表示するコマンドとしては使われません。

一般的にFreeBSDやMacでデフォルトゲートウェイを調べるには、**netstatコマンド**を用います。netstatコマンドは、現在のネットワーク接続状態を表示する際によく使われるコマンドですが、**-rオプション**を利用することで現在の経路テーブルを表示することができます。

次がnetstatコマンドで-rオプションを利用した場合の出力例です。なおここではIPv4アドレスのみを扱うものとし、IPv6アドレス(Internet6:以降)は省略しました。ホスト名で表示せずにIPアドレスで表示させるよう、**-nオプション**も同時に利用しています。

FreeBSD/Macのnetstatコマンド出力例

```
% netstat -nr
Routing tables

Internet:
Destination        Gateway            Flags      Refs      Use  Netif Expire
default            192.168.1.1        UGS           0        0   em0
192.168.1.0/24     link#1             U             0     1111   em0
127.0.0.1          link#5             UH            0       18   lo0

Internet6:
(省略)
```

上記には多くの項目がありますが、ここではDestinationとGatewayの値を利用します。Destinationは宛先ネットワークを指し、このサンプル例ではデフォルトゲートウェイを取得したいため"default"の行を読みます。この行のGatewayが192.168.1.1になっていますから、192.168.1.1がこのマシンのデフォルトゲートウェイです。

これでLinuxのrouteコマンドによるルーティングテーブル表示とほぼ同じ内容が得られたため、後は前項と同様に**awkコマンド**で抽出してpingを実行しています。

❶、❷、❸での処理は、P.146と同様ですので、そちらを参照してみてください。

注意事項

・前項の注意事項(→P.148)を参照してください。

関連項目

055 デフォルトゲートウェイにpingが通るかテストする(Linux)
114 サーバのping監視を行う

No. 057 pingで特定ホストへの応答平均時間を取得する

利用コマンド
ping, sed, awk

キーワード
ICMP, 応答速度, 平均値

いつ使うか
特定のサーバへの通信状態を調べたいとき

実行例

```
$ ./pingavg.sh
Ping to: 192.168.2.1
Ping count: 10
Ping average[ms]:
38.79
```

スクリプト

```
#!/bin/sh

ipaddr="192.168.2.1"
count=10

echo "Ping to: $ipaddr"
echo "Ping count: $count"
echo "Ping average[ms]:"

# pingコマンドを実行し、結果を一時ファイルに出力
ping -c $count $ipaddr > ping.$$

# "time=4.32 ms" の部分をsedで取り出し、awkで平均値を集計する
sed -n "s/^.*time=¥(.*¥) ms/¥1/p" ping.$$ |¥
awk '{sum+=$1} END{print sum/NR}'

# 一時ファイルを削除
rm -f ping.$$
```

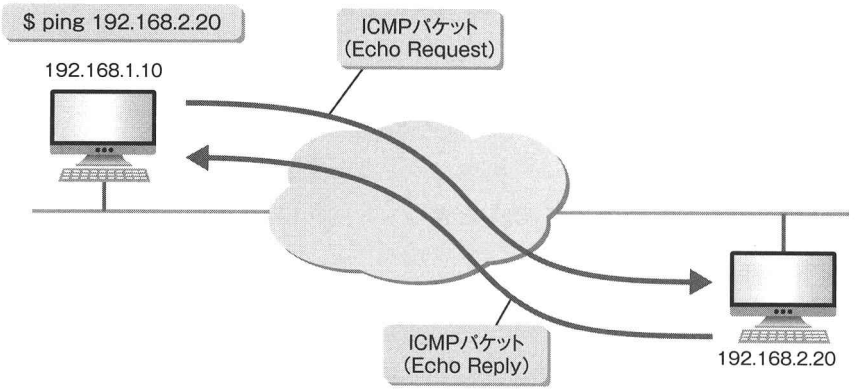
❶ ❷ ❸ ❹ ❺

解説

このスクリプトは、指定したIPアドレス宛てに複数回pingコマンドを実行し、応答時間の平均を計算して表示するものです。

pingコマンドは**ICMPパケット**を送信するコマンドです。実行すると、次のようにICMP echo requestパケットが宛先に送信され、その応答としてICMP echo replyパケットが返されます。

pingコマンドとICMPパケット



このICMPパケットの応答時間を測ることで、ネットワークの状態を調べることができます。例えば定期的にpingコマンドを実行しておき、急に応答時間が長くかかるようになったならば、そのホスト宛てのネットワークが混雑しているか、あるいは途中のネットワーク機器に何か異常が発生したのかもしれません。

pingコマンドの出力は、OSによって多少異なりますが、次のようになります。表示されているicmp_seqが試行回数、ttlがパケットのTTL値(ルータを何台超えられるか)、timeが応答時間(ミリ秒)です。

pingコマンドの出力例

```
$ ping -c 4 192.168.2.1
PING 192.168.2.1 (192.168.2.1) 56(84) bytes of data.
64 bytes from 192.168.2.1: icmp_seq=1 ttl=64 time=0.374 ms
64 bytes from 192.168.2.1: icmp_seq=2 ttl=64 time=0.405 ms
64 bytes from 192.168.2.1: icmp_seq=3 ttl=64 time=0.345 ms
64 bytes from 192.168.2.1: icmp_seq=4 ttl=64 time=0.469 ms

--- 192.168.2.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3000ms
rtt min/avg/max/mdev = 0.345/0.398/0.469/0.048 ms
```

最終行を見ると、rtt (round trip time, パケットが往復にかかった時間) のavgのところでpingコマンド自身が平均応答時間を出力してくれます。しかしここではコマンド出力から平均値を求めるサンプル例ということで、スクリプトで値を計算してみましょう。

❶で、対象のIPアドレスとpingコマンドの実行回数を指定しています。これを元に、❷でこれから計測する対象のIPアドレス、カウント数などを表示しています。

❸でpingコマンドを実行しています。pingコマンドはオプションを指定しないと永遠

に実行されるため、ここでは**-cオプション**で実行回数を指定しています。表示結果は後で計算処理するため、標準出力を一時ファイルping.$$にリダイレクトしています。この**$$**というのはシェルの特殊変数で、**プロセスID**が入っているため、このように一時ファイルを作る際のファイル名としてよく使われる手法です。詳しくはP.128を参照してください。

❹では、pingコマンドの出力結果から応答速度を取得しています。pingコマンドの応答速度は、「time=0.374 ms」の数字の部分です。msと付いているように、これはミリセカンド(ミリ秒)を意味します。

ここでは、**sedコマンド**の**-nオプション**と**pフラグ**でこの「time=0.374 ms」の部分をマッチさせ、数値部分のみを出力して応答時間を取り出しています。この、sedコマンドでパターンマッチした部分のみを出力する手法は、P.46で詳しく解説しています。なお、ここではパイプでawkコマンドに接続していますが、1行が長いため末尾を¥として改行しています。

sedコマンドで数値のみを切り出したため、❺でawkに渡される時点での入力は次のようになっています。これがpingコマンドの応答時間(ミリ秒)になります。

```
0.374
0.405
0.345
0.469
```

このような数値の羅列から平均値を求めるやり方はいろいろありますが、このサンプルでは**awkコマンド**を用いてみました。❺のようにawkのアクションでsumという変数に$1(1列目)の値を足していき、全行の和を求めます。最後にENDパターン内で、sumの値を処理した行数(NR)で割ることで、平均値を出力することができます。このNRとは、awkの組み込み変数で、入力中のレコード数を指します。

注意事項

- セキュリティ上の理由などから、最近はpingコマンド(ICMPパケット)に応答しないよう設定されたサーバも多く存在します。そのためこのサンプル例を利用する前に、まずは対象のサーバ宛てに手でpingを打ってみて、応答が返ってくるか確認してください。

関連項目

018 HTMLファイルから、タグの中に書かれたコマンドを抜き出してそのまま実行する
049 二重起動が可能な一時ファイル作成する
114 サーバのping監視を行う

No. 058 arpテーブルから指定IPアドレスに対応するMACアドレスを表示する

利用コマンド
arp, awk

キーワード
MACアドレス, IPアドレス, ARP

いつ使うか
ネットワーク内から、IPアドレスを指定して対象の物理アドレス(MACアドレス)を検索したいとき

実行例

```
$ ./arp.sh
192.168.2.1 -> 00:00:5e:XX:XX:XX
```

スクリプト

```
#!/bin/sh

ipaddr="192.168.2.1"

macaddr=$(arp -an | awk "/¥($ipaddr¥)/ {print ¥$4}") ―― ❶

if [ -n "$macaddr" ]; then ―― ❷
  echo "$ipaddr -> $macaddr" ―― ❸
else
  echo "$ipaddr は ARP キャッシュにありません" ―― ❹
fi
```

解説

このスクリプトは、シェル変数ipaddrで指定したIPアドレスを元に、OSの**ARPキャッシュ**を検索し、該当のネットワークインタフェースの**MACアドレス**を表示するものです。

ネットワークが予期せぬ動きをしているなどのトラブルシューティング時には、IPアドレスなどのネットワーク層だけではなく、その1つ下のレイヤであるデータリンク層での動作も調査すべきことがあります。ここではそのような際のツールとして使われる用途を考えています。

IPアドレスからMACアドレスを調べるには、システムのARPキャッシュを参照します。このARPキャッシュを操作するためのコマンドが、**arpコマンド**です。arpコマンドの出力フォーマットはOSによって多少違いますが、次のようになります。ここでは引数なし

の**-aオプション**を利用して、OSのARPキャッシュをすべて表示しています。また、**-nオプション**も付けて名前解決も行わないように指定しています。

arpコマンドの出力例

```
$ arp -an
? (192.168.2.2) at 74:8e:f8:XX:XX:XX [ether] on eth0
? (192.168.2.4) at d4:ae:52:XX:XX:XX [ether] on eth0
? (192.168.2.1) at 00:00:5e:XX:XX:XX [ether] on eth0
```

先頭の?はホスト名で、-nオプションで名前解決していないため全エントリが?となっています。次のカッコ内がIPアドレスで、そのIPアドレスに通信するためのMACアドレスが" at ...."と書かれています。

❶ではarpコマンドの出力を、awkを使って整形しています。まずawkのフィルタに/¥($ipaddr¥)/と、取得したいIPアドレスを指定しています。これにより該当IPアドレスの行だけをアクションで処理することができます。なお、上記の図を見るとわかるようにarpコマンドのIPアドレス前後にはカッコ()が付いているため、フィルタ指定もカッコ付きで書いています。

❶の**awk**のアクション{}の部分は、{print ¥$4}と、$4(4カラム目)を表示するように指定しています。この際、$4に対して¥でエスケープしています。これは、$ipaddrというシェル変数をawkコマンドの中に入れ込むために全体をダブルクォーテーションでくくっているため、$4とそのまま書くとシェルが展開しようとしてしまうためです。ここで$4はシェルの変数ではなく、「4カラム目」を意味するawkの変数です。awkコマンド内にシェルスクリプトからシェル変数を入れ込む際には、このように全体をダブルクォーテーションでくくってシェル変数を展開されるようにして、その代わりに$1、$2などのawk変数の$をエスケープするやり方が1つの手法です。

❶の結果でシェル変数macaddrに、検索したいIPアドレスに対するMACアドレスが代入できましたので、❷で値をチェックしています。ここでは**testコマンド**の**-n演算子**を用いて、空文字列でないかを調べています。空文字列でなければ**echoコマンド**で該当のIPアドレスとMACアドレスの組を表示し(❸)、空文字列ならばAPRキャッシュにないためその旨をechoコマンドで表示しています(❹)。

注意事項

- このサンプル例を試す際は、事前にARPキャッシュを作っておかなくてはいけません。具体的には、対象のマシンにpingを打つなどして、通信を発生させた直後に試してみてください。

No. 059 ホスト名からIPアドレスを取得する

利用コマンド

host, awk

キーワード

名前解決, DNS, IPアドレス, IPv4, IPv6

いつ使うか

DNSサーバへ問い合わせて名前解決を行い、該当のIPアドレスを一覧表示したいとき

実行例

```
$ ./hostip.sh
Address of www.google.com
=========
173.194.126.211 IPv4
173.194.126.208 IPv4
173.194.126.209 IPv4
173.194.126.210 IPv4
173.194.126.212 IPv4
2404:6800:4004:808::1011 IPv6
```

スクリプト

```
#!/bin/sh

# IPアドレスを取得したいホスト名を定義
fqdn="www.google.com"

echo "Address of $fqdn"
echo "========="

# hostコマンドでIPアドレスを取得し、awkで加工して出力
host $fqdn | ¥ ――――――――――――――――――――――――― ❶
awk '/has address/ {print $NF,"IPv4"} ¥ ─┐
/has IPv6 address/ {print $NF,"IPv6"}'  ─┘── ❷
```

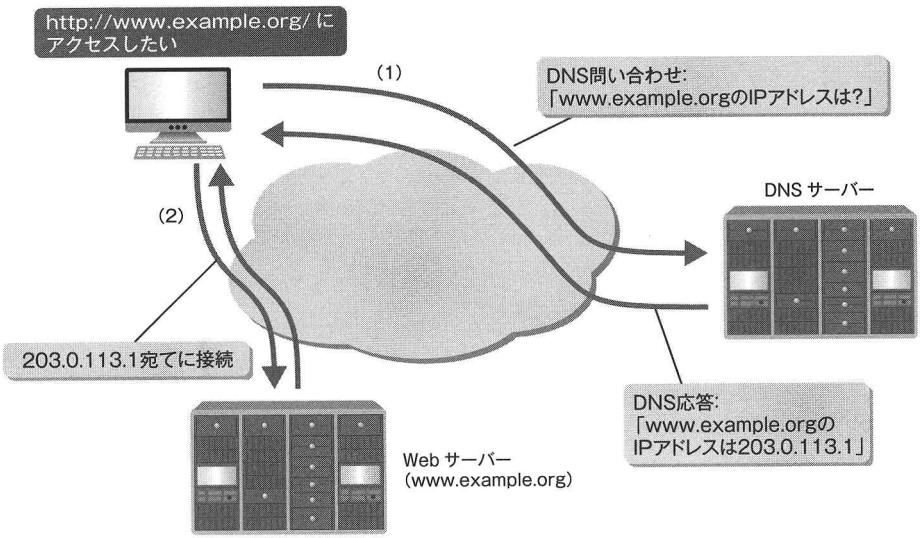
解説

このスクリプトは、hostコマンドを利用して指定したホストの名前解決を行い、IPアドレスを表示するものです。実行例ではwww.google.comのIPアドレスを表示してみました。なお、環境によってはhostコマンドが見つからずにエラーになる場合があります。その際は「注意事項」に従ってインストールを行ってください。

名前解決とは、ネットワーク上でホスト名からIPアドレスを取得することです。例えば普段使っているWebブラウザのアドレスバーにhttp://www.example.org/と入力すると、

コンピュータ内部ではまずDNSサーバに問い合わせてwww.example.orgの名前解決を行い、結果として203.0.113.1などのIPアドレスを取得してそのアドレス宛てに通信を行います。

DNSと名前解決



このように名前解決とは、地味ながらも非常に重要なネットワーク通信の要素です。IPアドレスでは通信できるのにホスト名で指定すると接続できない、という場合は名前解決がうまくいっていないパターンが多いでしょう。そのような際のトラブルシューティングのツールとして、このスクリプトが役立つかもしれません。

シェルスクリプトからDNSサーバに問い合わせて名前解決を行うには、**nslookupコマンド**や**digコマンド**などを使う方法がありますが、このサンプル例では表示がシンプルな**hostコマンド**を利用してみました。例えばwww.google.comへのhostコマンドの出力結果は、次のようになります。

hostコマンドの実行例

```
$ host www.google.com
www.google.com has address 74.125.235.84
www.google.com has address 74.125.235.80
www.google.com has address 74.125.235.83
www.google.com has address 74.125.235.82
www.google.com has address 74.125.235.81
www.google.com has IPv6 address 2404:6800:4004:808::1013
```

大規模なサイトは負荷分散のため、同一の**FQDN**（完全修飾ドメイン名：Fully Qualified Domain Name）に複数のIPアドレスを持つことが多く、上記でもwww.google.comに対して複数のIPアドレスが表示されています。またhostコマンドでは、対象FQDNにIPv6アドレスもある際は、「www.example.org has IPv6 address……」としてIPv6アドレスがIPv4アドレスと同時に表示されます。

サンプル例では、❶で、まずシェル変数fqdnに指定されたホストの名前解決を行うため、hostコマンドを実行します。この結果をパイプでつないでawkで処理するのですが、長くなってしまうため行末に¥を付けて改行を入れています。

❷の**awkコマンド**では、パターンを2つ設定しています。1つ目のパターンは/has address/で、これはIPv4アドレスを取得します。このパターンにマッチする行は、次のように最後にIPアドレスが入っています。

```
www.example.org has address 203.0.113.1
```

awkで最後のカラムを取り出すには、最終カラムを表す変数NFを用いてprint $NFと書きます。ここではさらに、後ろに"IPv4"とスペース区切りで表示しています。

awkコマンドの2つ目のパターンは/has IPv6 address/で、これはIPv6アドレスを取得します。IPv4アドレスのときと同様に、最終カラムにIPアドレスが入っていますから、これをawkの変数NFから取り出して出力しているわけです。

注意事項

- CentOSでは、最小構成でインストールするとhostコマンドがインストールされていません。そのため次のようにyumコマンドでbind-utilsパッケージをインストールしてください。正しくインストールできたかどうかは、whichコマンドで確認できます。

bind-utilsパッケージに含まれるhostコマンドをインストール（CentOSの場合）

```
$ su
Password:                    パスワードを入力
# yum install bind-utils
# exit
$ which host
/usr/bin/host
```

- Ubuntuの場合はhostコマンドはデフォルトでインストールされます。もし存在しない場合はapt-getコマンドでbind9-utilsパッケージをインストールしてください。
- hostコマンドはDNSサーバへ問い合わせて名前解決するコマンドです。そのため、/etc/hostsに書いた設定内容はこのスクリプトでは無視されます。

No. 060 IPアドレスからホスト名を逆引きする

利用コマンド
host, awk, sed

キーワード
IPアドレス, ホスト名, 逆引き, DNS

いつ使うか
IPアドレスが書かれたファイルを読み込み、ホスト名を付けて表示したいとき

実行例

```
$ cat ip.txt
198.51.100.43
203.0.113.1
203.0.113.198
$ ./revlookup.sh ip.txt
198.51.100.43,www.example.org
203.0.113.1,mail.example.com
203.0.113.198,
```

IPアドレスが記述されたファイル

スクリプト

```
#!/bin/sh

while read ipaddr ―――――――――――――――――❶
do
  # hostコマンドでIPアドレスを逆引きする
  revlookup=$(host "$ipaddr") ―――――――――❷

  # hostコマンドの逆引きが成功したかどうか
  if [ $? -eq 0 ]; then ―――――――――――――❸
    echo -n "$ipaddr," ―――――――――――――❹
    # hostコマンドの出力をawkで整形してホスト名だけ表示
    echo "$revlookup" | awk '{print $NF}' | sed 's/¥.$//' ―――❺
  else
    echo "$ipaddr,"
  fi

  # DNSサーバへの負荷軽減のため1秒のウェイト
  sleep 1 ―――――――――――――――――――――❻
done < $1
```

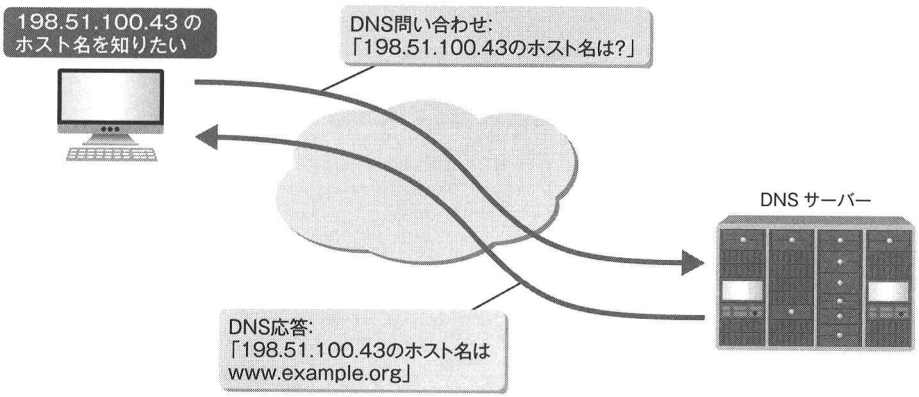
解説

このスクリプトは、IPアドレスが書かれたファイル(ip.txt)を読み込んで、それぞれのIPアドレスをhostコマンドによって逆引きし、「IPアドレス,ホスト名」の組でCSVファイルとして出力するものです。

Apacheなどのサーバソフトウェアのログファイルには、接続元IPアドレスが記録されています。これらのログを分析する際、IPアドレスを逆引きすると接続元のISPや回線などがわかるため、接続元IPアドレスをホスト名に変換したい場合がよくあるでしょう。このスクリプトはそのようなケースに利用できます。

なおIPアドレスの**逆引き**とは、次のようにIPアドレスをDNSサーバに問い合わせ、ホスト名を取得するものです。

DNSの逆引きとは



通常、DNSで**名前解決**を行う場合は、P.157で見たようにホスト名からIPアドレスを取得します。このケースでは逆にIPアドレスからホスト名を取得するので、「逆」引きと呼ばれるのです。なお、文書によってはホスト名から名前解決を行うことを「正引き」と呼ぶこともあります。

サンプル例では、まず❶のwhile文で**readコマンド**を実行することで、ファイルからシェル変数ipaddrにIPアドレスを読み込んでいます。ここでは、次のようなIPアドレスが羅列されたファイルを読み込むものと仮定します。

```
198.51.100.43
203.0.113.1
203.0.113.198
```

またこのwhile文の最終行(done)の後に、コマンドライン引数$1を入力リダイレクトしています。そのため、readコマンドを用いてシェル変数ipaddrにファイルの内容が1行

ずつ読み込まれていきます。

❷で、IPアドレスからホスト名を逆引きするために**hostコマンド**(→P.157)を実行して、その結果をコマンド置換$()にてシェル変数revlookupに代入しています。

❸で、hostコマンドの終了ステータスを特殊変数$?で判断しています。hostコマンドで正常に逆引きできた場合、終了ステータスである**$?**は0(正常終了)となっています。一方、逆引きに失敗した場合は終了ステータスは0以外となるため、ここではこの$?の値を用いて、逆引きできたかどうかをif文で分岐しています。

正常に逆引きできていた場合は、[IPアドレス,ホスト名]を出力するために、まず**echoコマンド**を改行なしの**-nオプション**で実行し、IPアドレスを表示します(❹)。なおこのechoコマンドの-nオプションの例はMacではエラーとなりますので、P.15を参照してください。

続いて❺でホスト名の表示を行います。次の結果を見るとわかるとおり、逆引きできたホスト名はhostコマンドの表示結果の最後にあるため、ここでは**awkコマンド**を用いて最終カラム**$NF**を出力しています。NFとは、awkコマンドで「最後のカラム」を意味する変数です。

hostコマンドによる逆引きの出力例

```
$ host 198.51.100.43
43.100.51.198.in-addr.arpa domain name pointer www.example.org.
```

なお、hostコマンドの出力では、ホスト名の最後に.(ドット)が付いています。そのため❺ではawkコマンドをさらにパイプでつないで**sedコマンド**に渡し、sedコマンドで行末の.(ドット)を消しています。ここでは、行末のドットを¥.$というパターンでマッチさせて、これを空文字列に置換することで行末ドットを削除しています。

最後に❻で、1秒のウエイトを入れています。これはDNSサーバに負荷をかけないようにするための処置です。hostコマンドは外部のDNSサーバに問い合わせをするコマンドですから、このサンプル例のようにIPアドレスをファイルから読み込んで順次処理する場合、ファイルの行数が多ければ大量のDNS問い合わせをサーバへと投げることになってしまいます。そのような行為はサーバに負荷をかけてしまいますから、ここでは1行ごとに1秒ウェイトを入れて、DNSサーバへの負荷を抑えています。

注意事項

- このサンプルの中で例示しているIPアドレスは、例示用IPアドレスとしてRFC 6890で定義されているもので、実在のIPアドレスではないため逆引きには失敗します。そのため、このサンプルを試す際は実在するIPアドレスに置き換えてください。

関連項目

059 ホスト名からIPアドレスを取得する

No. 061 あるサーバの特定ポートへ通信できるかのチェックスクリプト

利用コマンド

nc

キーワード

ネットワーク，ポート番号，ポートスキャン，ファイアウォール，テスト

いつ使うか

ネットワーク・ファイアウォールの設定が正しいかどうかのテストを、スクリプトで実行したいとき

実行例

```
$ ./checkport.sh
Connection to 192.168.2.52 80 port [tcp/http] succeeded!
Connection to 192.168.2.52 8080 port [tcp/webcache] succeeded!
$ cat fail-port.log
Failed at port: 2222 ――― 接続に失敗したポート番号がログに出力されている
```

スクリプト

```
#!/bin/sh

ipaddr="192.168.2.52"
faillog="fail-port.log"

# テストするポートは80,2222,8080
for port in 80 2222 8080 ―――――――――――――――――――――――― ❶
do
  nc -w 5 -z $ipaddr $port ―――――――――――――――――――――― ❷
  if [ $? -ne 0 ]; then
    echo "Failed at port: $port" >> "$faillog" ――――― ❸
  fi
done
```

解説

このスクリプトは、**ncコマンド**を用いて対象サーバのTCPポートの状態を調べ、接続できないTCPポート番号をfail-port.logファイルに出力するものです。

サーバ構築の際には、ネットワークやファイアウォールの設定が正しいかどうかのテストが必要になる場面が多々あります。そのようなときに、ひとつひとつのポートを手でテストするのではなく、テストが必要なポートをまとめてチェックしてくれるこのようなスクリプトを用意しておくと便利でしょう。また、スクリプトにしておけばテスト手順の漏れもなくせますし、自動化できるという利点もあります。

まずテストするTCPポート番号の定義として、❶で、for文を利用してシェル変数portに80,2222,8080という数値を順に代入しています。このサンプル例では、この数値が実際に接続テストを行うポート番号になります。

❷で接続テストを行うため、ncコマンドを実行しています。ncコマンドはNetcatと呼ばれ、TCP/UDPパケットの作成を行いさまざまなネットワークテストができます。そのためncコマンドは、ネットワーク技術者によく使われるコマンドです。ncコマンドにはさまざまな機能がありますが、基本的な使い方は次のようになります。

書式 ncコマンドの書式

nc *[オプション] <対象ホスト> <ポート番号>*

ここでは、ncコマンドの-wオプションと-zオプションを利用しています。

-zオプションはTCPの3WAYハンドシェイクのみを行い、実際のデータ通信は行わないオプションです。このサンプル例のように、ネットワーク/ファイアウォールのチェックを行う目的で、サーバまでの疎通だけを確認したい場合には、この-zオプションが便利です。

-wオプションはタイムアウト秒数の設定です。ファイアウォールの設定によっては、サーバの存在自体を隠すために、ポートへの接続時に何も応答せずに無視する機能を利用していることがあります。このような場合にはncコマンドの動作が停止してしまいますから、このサンプル例では-w 5として5秒応答がなければタイムアウトで終了するように処理しています。

ncコマンドでよく使うオプションを次にあげておきました。

ncコマンドの主要オプション

オプション	説明
-k	コマンドを終了させず、Ctrl + Cが入力されるまで永続化する
-l	リッスンモードで起動する
-n	ポート番号のサービス名への変換や、ホスト名の名前解決を行わない
-u	TCPパケットではなくUDPパケットを送信する
-v	verboseモード。通信状態の出力を細かく行う
-w	タイムアウト秒数を設定する
-z	データ通信は行わずハンドシェイクのみを行う

ncコマンドでは、接続に失敗した場合は終了ステータスに非ゼロを返します。シェルスクリプトでは直前のコマンドの終了ステータスは変数$?で取得できるため、❸のif文でncコマンドが成功したか失敗したかを判断しています。**-ne**は「等しくないならば真」ですから、このif文の中は非ゼロの場合、すなわちncコマンドが失敗した場合に実行されます。こうして、失敗したポート番号のみをfail-port.logに出力することができるのです。

Netcatについて

本項で利用したNetcatと呼ばれるコマンドには、このサンプル例で紹介したncコマンドのほかに、ポートスキャンツールnmapに付属する**ncatコマンド**があります。また、GNUによるGNU Netcatというソフトウェアもあります。ただしGNU Netcatは既に開発が止まっているため、現在はほとんど使われていません。一方、ncコマンドとncatコマンドはどちらも頻繁に利用されています。

ncコマンドとncatコマンドは、オプションもほぼ同じで似た動きをしますが、細かな動作やメッセージ出力に差異があります。そのためシェルスクリプトで利用する際には注意が必要です。一般に「Netcatコマンド」と言うときには、ncコマンドを指している場合とncatコマンドを指している場合がありますので、事前にどちらのNetcatなのかきちんと確認したほうがよいでしょう。

注意事項

- 他人のサーバのポートを不用意にチェックすると、ポートスキャンとして不正アクセス行為とみなされる可能性があります。そのためこのサンプルを試す際は、自分の管理下にあるサーバのみに行うよう注意してください。
- ncコマンドの-wオプションでのタイムアウト設定は、Macの場合にはマニュアルには記載されていますが動作しません。そのためMacで応答がない場合は、Ctrl + Cで終了させてください。

関連項目

062 テスト用の簡易TCPサーバを立ち上げる

No. 062 テスト用の簡易TCPサーバを立ち上げる

利用コマンド

nc

キーワード

ネットワーク，ポート番号，ファイアウォール，テスト，デーモン

いつ使うか

システム構築時、ミドルウェアなどをまだインストールしていないサーバに対して、ネットワーク疎通のテストを行いたいとき

実行例

```
$ ./port-httpd.sh
Connection from 192.168.2.5 port 8080 [tcp/webcache] accepted
```

起動した状態で、別のサーバからncコマンドでパケットを投げる

スクリプト

```
#!/bin/sh

port=8080
nc -v -k -l $port ―――――――――――――――――――――❶
```

解説

このスクリプトは、サーバ上のシェル変数portで指定されたポート番号で、TCP接続を受け付けるものです。ネットワークの疎通を調べる際に使われるケースを想定しています。

ncコマンドは、TCP/UDPパケットの作成を行い、さまざまなテストができます。サンプル061では、クライアント側から送信するパケットを作成するために使いました。ncコマンドは多様な機能を持ち、このサンプル例のようにサーバ側で待ち受けて、簡単なデーモンとして振る舞うことも可能です。ここではこのデーモンとして振る舞う機能を、ネットワークのテスト用スクリプトとして利用します。

❶でncコマンドを、3つのオプション-vと-k、-lを用いて利用しています。

-vオプションは、verboseモード(冗長モード)です。接続があった際にメッセージを表示してくれますので、テスト用ではわかりやすいため付けておいたほうがよいでしょう。

-lオプションは、リッスンモードで起動する指定です。これにより、サンプル例ではポート8080で待ち受けるプロセスが起動できます。

-kオプションは、永続化オプションです。ncコマンドのリッスンモードは、通常は一度接続を受け付けるとそこでncコマンドが終了してしまいます。ネットワークテスト時に

は何度もパケットを投げて確認するでしょうから、そのたびに終了しては困ります。そのため、コネクションを永続化する-kオプションを付けることで、何度接続を受け付けても終了しないようにできます。なおこの際は、キーボードから Ctrl + C を入力することで終了できます。

ncコマンドのオプションについてはP.163も参照してください。

このサンプル例の実際の利用シーンとしては、以下のようになります。

1) ネットワークの疎通を確認したい接続先のサーバで、このサンプル例のport-httpd.shを起動する。
2) ネットワークの疎通を確認したい接続元のサーバで、P.162のスクリプトでパケットを生成する。
3) ネットワーク状態のOK/NGを判断する。

特にシステム構築時に、物理サーバだけはすでにあるが、まだミドルウェアのインストールや設定は行っていない。しかしネットワーク担当は早く設定作業にとりかかりたいとせっついている……というシーンはよく見られます。そのような際には、このサンプル例のような簡易デーモンを起動して疎通試験を行う、という例がよく見られます。

単にncコマンドを実行するだけならばわざわざスクリプトにしておく必要もなさそうですが、このようにシェルスクリプトにしておくと、テストを実行する際にコマンド操作が不慣れな人でも比較的楽に操作することができます。そのため手順書などを簡略化できますし、作業時のミス(ポート番号の指定を誤る、など)をなくすという意味でも重要です。このようなテストの作業は、できるだけ自動化できるようスクリプト化することを検討してみてください。

注意事項

- Macでは、ncコマンドの-vオプション(verboseモード)はマニュアルには記載されていますが、-vオプションを利用しても接続があった際のメッセージは表示されません。
- ポート番号1024未満のスタンダードポート(ウェルノウンポート)にバインドするにはroot権限が必要なため、1024未満のポート番号でテストする際は、このスクリプトもroot権限で実行する必要があります。

関連項目

061 あるサーバの特定ポートへ通信できるかのチェックスクリプト

No. 063 ftpで自動ダウンロード・自動アップロードを行う

利用コマンド

ftp

キーワード

ftp, ログイン, 自動化

いつ使うか

ftpでファイル連携を行うシステムで、ログイン処理やファイルのダウンロード・アップロードを自動化したいとき

実行例

```
$ ./autoftp.sh
$ ls
autoftp.sh  app.log ―――――――― app.logがダウンロードできた
```

スクリプト

```
#!/bin/sh

# FTP接続のための設定
server="192.168.2.5"
user="user1"
password="xxxxxxxxx" ―――――――― ❶
dir="/home/user1/myapp/log"
filename="app.log"

ftp -n "$server" << __EOT__ ―――――― ❷
user "$user" "$password"
binary
cd "$dir"
get "$filename"
__EOT__
```

解説

このスクリプトは、指定されたFTPサーバから、**ftpコマンド**で自動的にファイルをダウンロードするためのものです。スクリプト内のシェル変数で、接続するFTPサーバやアカウントを指定すると、自動的にログインしてファイルダウンロードが行えます。

ftpはファイル転送のためのプロトコルで、サーバ間でファイルをやり取りするためによく使われています。このftpは古いプロトコルであり、平文でID・パスワードをやりとりするというセキュリティ面での欠点を持つため、新規システムで採用されることは減りつつあるようです。しかし歴史の長いプロトコルであることから、読者がシステム運用を行う上でFTPサーバを扱うことになるケースもまだまだ多いでしょう。

ここでは、バックアップサーバ上で定期的にスクリプトを実行して、FTPサーバ兼アプリケーションサーバのログファイルapp.logを取得するというケースを想定しています。このようにログファイルなどを定期的にftpで取得したいという要件は、現在でも比較的よく見られます。

まずはじめに❶で、ftp接続のための各種設定をシェル変数に代入しています。

- server　：FTPサーバのIPアドレスもしくはホスト名
- user　：FTPサーバへのログインID
- password：FTPサーバへのログインパスワード
- dir　：FTPサーバ上のファイル設置ディレクトリ
- filename　：ダウンロードするファイル名

続いて❷で、ftpコマンドを実行します。ftpコマンドはオプションなしで実行すると、次のように対話型モードで起動されます。これは、人がキーボードを打つ際にはわかりやすいのですが、シェルスクリプトとして自動実行させようとすると、このような対話型というのはむしろ扱いにくいモードになります。

ftpコマンドをオプションなしで実行すると対話型モードになる

```
$ ftp 192.168.2.5
Connected to 192.168.2.5 (192.168.2.5).
220 ftpserver FTP server (Version 6.00LS) ready.
Name (192.168.2.5:user1): user1
331 Password required for user1.
Password: ──────────────────────パスワード入力
230 User user1 logged in.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd /home/user1/ftp
250 CWD command successful.
```

そこで、シェルスクリプトとして自動実行したいケースでは、❷のようにftpコマンドの-nオプションを利用するのがよく使われる手法です。**-nオプション**とは、本来は.netrcファイルによる自動ログインを抑止するオプションですが、シェルスクリプトでの自動化のためにもよく使われるオプションです。

ftpコマンドでは、ホームディレクトリ配下の.netrcというファイルにログイン情報を記載しておくと、自動ログインが行えます。ここで-nオプションを用いると、.netrcファイルを利用せず、代わりに標準入力からftpのコマンドを与えることができるようになります。サンプル例ではこの機能を利用し、❷で**ヒアドキュメント**の形でftpコマンドを自動実行させています。なお、シェルスクリプトでのヒアドキュメントの書き方については、P.56で解説していますのでそちらを参照してください。

❷のヒアドキュメントで実行しているftpコマンドは、具体的には以下のものです。

1) userコマンドでログインを行う
2) binaryコマンドでバイナリモードを設定
3) cdコマンドでディレクトリ移動
4) getコマンドでファイル取得

こうしてftpコマンドの-nオプションを利用すれば、ftpプロトコルによるファイルダウンロードを、シェルスクリプトで自動化することができます。なお、サンプル例のヒアドキュメント内でgetしている部分をputに変えれば、ファイルのアップロードを自動化することもできます。

注意事項

- CentOSでは、デフォルトではftpコマンドがインストールされていません。そのため次のようにyumコマンドでインストールを行ってください。

```
# yum install ftp
```

- Linuxのftpコマンドは、-nオプションを付けると、実行例のようにFTPサーバとのやり取りが一切表示されなくなります。
- FreeBSDやMacのftpコマンドは、-nオプションを付けていてもftpサーバとのやり取りが表示されます。これがうるさく感じる場合、FreeBSDやMacではftpコマンドの-Vオプションを併用して表示を抑制できます。
- このシェルスクリプトにはftp接続のためのIDとパスワードが直接書かれているため、他人から見られないようにする必要があります。ファイルのパーミッションを700にするなど適切な設定を行ってください。
- ftpは平文で通信を行うため、通信内容を傍受されるとIDとパスワードが簡単にわかってしまいます。そのため、このスクリプトは基本的にイントラネット内で利用し、インターネット経由でのファイルのやり取りには、scpやsftpなどの暗号化通信できるプロトコルを選択するようにしてください。

関連項目

022 ヒアドキュメントで変数展開をせずにそのまま$strのように表示する

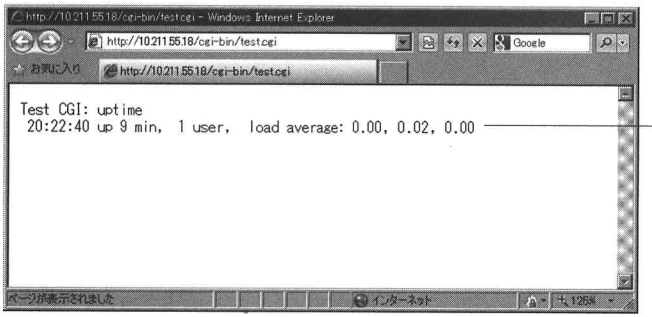
No. 064 シェルスクリプトでCGIを実行する

利用コマンド
echo, uptime

キーワード
CGI

いつ使うか
シェルスクリプトでCGIプログラムを記述したいとき

実行例



ファイルを/cgi-bin/test.cgiとして設置し、Webブラウザでアクセスする

スクリプト

```
#!/bin/sh

# CGI ヘッダの出力
echo "Content-Type: text/plain" ―――❶
echo

# コマンドを実行してブラウザに表示させる
echo "Test CGI: uptime" ―――――――❷
uptime
```

解説

このスクリプトは、CGIプログラムとしてシェルスクリプトを実行するためのものです。Webブラウザでアクセスすると、**uptimeコマンド**の出力結果を表示し、サーバのロードアベレージなどを確認できます。

CGIプログラムを記述する際、プログラミング言語として一般的にはPerlやRubyがよく使われます。そのため、シェルスクリプトでCGIプログラムを書くというのは奇妙なやり方と思われるかもしれません。しかし、CGIプログラムはPerlで書かなければならないというものではなく、Webブラウザが解釈できるように適切に出力をすることができれ

ば、利用する言語は何でもかまいません。

特にサンプル例のように、コマンドの出力結果をちょっとWebブラウザで表示したいというケースでは、シェルスクリプトで書いたほうが便利でしょう。

❶で、CGIのヘッダを出力しています。このサンプルではコマンドの出力結果をそのままブラウザで表示したいため、text/plainで出力することにします。また、HTTPのヘッダ部とボディ部は空行で区切る必要があるため、続けて空のechoコマンドを用いて改行しています。

これらCGIヘッダなどについての詳しい説明は本書の範囲を超えますので、興味のある方はCGIプログラミングの書籍などを読んでみてください。

❷がHTTPレスポンスのボディ部、つまりWebブラウザに表示される部分を出力している箇所です。"Test CGI: uptime"という文字列を**echoコマンド**で表示した後に、uptimeコマンドを実行しています。uptimeコマンドとは、次のようにサーバの起動時間と、過去1分/5分/15分のロードアベレージを表示するコマンドです。

uptimeコマンドで起動時間やロードアベレージを表示

```
$ uptime
 23:34:55 up  3:21,  1 user,  load average: 0.74, 0.40, 0.40
```

こうしてCGIとしてシェルスクリプトを用いることで、Webブラウザから実行結果を確認できます。

この他にシェルスクリプトでCGIを書く利用例として、サーバ構築時のテスト用途があげられます。例えば最小構成のFreeBSDやLinux (CentOS) は、インストール直後のデフォルト状態ではPerlがインストールされていません。サーバの構築作業とプログラム開発が並行しているようなプロジェクトでは、まず臨時にWebサーバとして動作させ、クライアントからCGIに接続できるかどうかのテストを先に行いたいという場面がよくあります。

このようなケースでは、シェルスクリプトで書いたCGIプログラムを設置して、クライアントからの接続テストを先行して進めるという手法をとることができるでしょう。

注意事項

- サーバに対して負荷の高いコマンドを実行するCGIプログラムを、インターネットからもアクセスできるところに公開すると、大量のアクセスを受けてサーバの動作に影響を与える可能性があります。そのため、そのようなCGIプログラムは、イントラネット内にのみ公開する、パスワードによる保護をかけるなど、アクセス制限をかけることを検討してください。

No. 065 指定したサイズのファイルを作り、転送速度を測定する

利用コマンド

dd, time, ftp, bc

キーワード

転送速度, 通信速度, ネットワーク速度

いつ使うか

あるサーバ宛てに、一時ファイルを転送して通信速度を測りたいとき

実行例

```
$ ./transfer-sec.sh
Filesize: 1024(KB)
FTP Server: 192.168.2.5
Transfer Speed: 978 (KB/sec)
```

スクリプト

```
#!/bin/sh

# 転送速度を測る一時ファイルのサイズ指定。単位はキロバイト (KB)
filesize=1024 ―――❶
# 転送速度を測る一時ファイルのファイル名 (テンポラリ)
tmpdata="tmpdata.tmp" ―――❷
timefile="timecount.tmp"

# 転送に用いる一時ファイルを作成する
dd if=/dev/zero of="$tmpdata" count=$filesize bs=1024 2> /dev/null ―――❸

# FTP接続してファイルをPUTする (No.063参照)
server="192.168.2.5"   ┐
user="user1"           ├―――❹
password="xxxxxxxxx"   ┘

echo "Filesize: $filesize (KB)"
echo "FTP Server: $server"

(time -p ftp -n "$server") << __EOT__ 2> "$timefile" ―――❺
time ftp -n "$server" << __EOT__
user "$user" "$password"
binary
put "$tmpdata"
__EOT__

# time コマンドの出力結果から実時間を取得し、割り算を
```

```
# 行って速度を求める
realtime=$(awk '/^real / {print $2}' "$timefile") ―――❻
speed=$(echo "${filesize}/${realtime}" | bc) ―――❼
echo "Transfer Speed: $speed (KB/sec)"

# 転送一時ファイルの削除
rm -f "$tmpdata" "$timefile"
```

解説

このスクリプトは、一時ファイルを作成してFTP転送し、その転送速度を測定するものです。このスクリプトを起動したサーバと、シェル変数serverで指定されたサーバ間の、通信速度を調べる用途での利用を想定しています。

このサンプル例のように、サーバ間でファイルのやり取りを実際に行ってみて転送速度を測ることは、ネットワークの運用において重要な作業です。そこでこのようなシェルスクリプトを利用して、できるだけ作業を自動化してみましょう。

サンプルは、まず❶と❷で、速度測定するための一時ファイルの、ファイルサイズおよびファイル名(tmpdata)を設定しています。同時に、時間測定のための一時ファイル名(timefile)も設定しています。

ここではファイルサイズは1024KB(1MB)を指定していますが、ネットワークの状態によって適正な値は異なるでしょう。あまりサイズが小さすぎると転送が一瞬で終わってしまって正確な速度を計算できませんし、サイズが大きすぎるとサーバやネットワークに大きな負荷をかけてしまいます。読者の環境にあわせて、ファイルサイズは適宜設定してみてください。

❸で、転送する一時ファイルを**ddコマンド**で作成しています。ddコマンドは指定された入力から出力へコピーを行うコマンドです。このスクリプトではddコマンドの経過出力は表示する必要がないため、**/dev/null**へと標準エラー出力をリダイレクトしています。ここでは以下のようにオプションを指定しています。

一時ファイル作成に用いたオプション

オプション	説明
if=/dev/zero	入力に、/dev/zeroというヌルキャラクタが読み出せるスペシャルデバイスを指定
of="$filename"	出力に、シェル変数filenameで指定されたファイル名を指定
count=$filesize	シェル変数filesizeで指定した回数だけコピーを行う
bs=1024	コピーを行うブロックサイズ。ここでは1024バイト(1キロバイト)

これで、**ヌルキャラクタ**(ASCIIコードで0x00)で埋められた、1024(KB)×1024(回)＝1MBの一時ファイルができあがります。

なお、このように**/dev/zero**から一時ファイルを作ると高速にファイルを作成できますが、中身はすべて同じデータのファイルができます。転送速度を測る際、プロトコルに

よっては自動的にファイル圧縮がかかるため、このように内容がすべて同じファイルは圧縮率が非常に高くなってしまいあまり現実的ではありません。

そのため用途によっては/dev/zeroからではなく**/dev/urandom**から読み込むようにすると、ファイル内容をランダムにした、より現実的なファイルが作れます。

```
dd if=/dev/urandom of=tmp.dat count=1024 bs=1024
```

なおサンプル例ではftpプロトコルを利用しています。これは自動的にファイル圧縮がかかるプロトコルではないため、高速性を重視して/dev/zeroからファイルを作っています。

転送ファイルが用意できたので、続いて実際にファイルを送信します。❹では、ftpでファイル転送する設定を記述しています。このようにシェルスクリプト中でftp転送を自動で行う方法は、P.167で詳しく説明していますのでそちらを参照してください。

ftp転送の際、❺で**timeコマンド**を利用しています。ここでは秒数のみを表示する-pオプションを利用して、実時間realを取得します。timeコマンドでの実行時間の取得については、P.253を参照してください。

❺は**ヒアドキュメント**とリダイレクトが入り交じっていてちょっとわかりにくいのですが、"__EOT__"というヒアドキュメントを用いて、同時にtimeコマンドの出力をシェル変数timefileで指定されているファイルにリダイレクトしています。

また、❺はコマンド全体を丸カッコ()でくくって**サブシェル**としています。これはコマンド全体をひとくくりにしてリダイレクトするための処理です。サブシェルを使わずに次のように書くと、ftpコマンドの結果のみがリダイレクトされてしまい、timeコマンドの結果が得られません。

```
time -p ftp -n "$server" << __EOT__ 2> "$timefile"
```

❻で、転送速度を計算しています。まずtimeコマンドの表示結果が出力されているファイル$timefileから、コマンド実行にかかった実時間realを取得します。

転送速度は、「ファイルサイズ÷時間」ですから、この計算を❼の**bcコマンド**で行っています。シェルスクリプトの計算ではexprコマンドがよく使われますが、**exprコマンドは小数が扱えない**ため、ここではbcコマンドを用いています。割り算の式をechoコマンドで表示し、それをパイプでbcコマンドに入力することで計算結果が得られます。

このサンプル例では、ファイルサイズは単位がキロバイト、timeコマンドの出力は秒数でしたから、❼で計算している速度の単位は「キロバイト／秒」となります。これをechoコマンドで表示して転送速度が表示できるわけです。

注意事項

・P.167で、ftpはセキュリティ的な問題があると記載しましたが、ここでは敢えてftpを用いました。これは、scpなどのsshベースの通信は、暗号化によるCPUオーバーヘッドが

大きく、通信速度を測ってもあまり意味のある値が得られないからです。ここではサーバ間でのファイル転送速度を純粋にとりたいため、ftpを用いています。

- このスクリプトで測る時間は、純粋なネットワークの通信速度だけではなく、一時ファイルのディスク読み書きぶんにかかる時間も入っています。そのため、ディスク速度がとても遅いサーバの場合は、見かけの速度が遅くでるでしょう。
- wgetコマンドやcurlコマンドなどのツールは、コマンド終了時に転送速度を表示してくれます。そのためダウンロードの速度を見たい場合は、これらのツールの出力を確認してもよいでしょう。

wgetコマンドでのダウンロード

```
$ wget "ftp://user1:xxxxxxx@192.168.2.5/tmpdata.tmp"
... (省略) ...
2014-03-15 09:21:11 (2.9 MB/s) - "tmpdata.tmp" saved [44442068]
```

curlコマンドでのダウンロード

```
$ curl -u "user1:xxxxxxx" -O "ftp://192.168.2.5/tmpdata.tmp"
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 42.3M  100 42.3M    0     0  5.3M      0  0:00:08  0:00:08 --:--:--  5.9M
```

なお、この際、FTPのIDとパスワードは次のように指定します。

```
#wgetコマンド
wget "ftp://<ユーザ名>:<パスワード>@<ホスト名>/<ファイルパス>"
#curlコマンド
curl -u "<ユーザ名>:<パスワード>" -O "ftp://<ホスト名>/<ファイルパス>"
```

関連項目

063 ftpで自動ダウンロード・自動アップロードを行う
091 scpでファイル転送を行ってCPU利用率を計算し、圧縮処理をすべきかどうか判断する

No. 066 IPアドレスによる処理分岐をcase文で書く

利用コマンド
case, ping

キーワード
IPアドレス, 分岐, マッチング, 文字列

いつ使うか
IPアドレスごとに違う処理を行う際、簡単に分岐を書きたいとき

実行例

```
$ ./ipmatch.sh 192.168.2.1
Ping to 192.168.2.1 : [OK]
$ ./ipmatch.sh 192.168.3.1
192.168.3.1 はテスト対象ではありません
```

スクリプト

```
#!/bin/sh

# 対象IPアドレスがコマンドライン引数で指定されていなければ
# エラーとして終了
if [ -z "$1" ]; then
  echo "IPアドレスを指定してください" >&2
  exit 1
fi

# テスト対象ネットワークならばpingコマンドを実行する
case "$1" in
  192.168.2.*|192.168.10.*)
    ping -c 1 "$1" > /dev/null 2>&1

    if [ $? -eq 0 ]; then
      echo "Ping to $1 : [OK]"
    else
      echo "Ping to $1 : [NG]"
    fi
  ;;
  *)
    echo "$1 はテスト対象ではありません" >&2
    exit 2
  ;;
esac
```

❶ ❷ ❸ ❹ ❺

解説

このスクリプトは、引数で指定されたIPアドレス宛てに**pingコマンド**で**ICMPパケット**を送り、ネットワーク疎通を確認するものです。この際、テストとは関係のないネットワークにはpingコマンドを実行しないようにする処理を入れています。なお、ここでは「192.168.2.0/24および192.168.10.0/24」のみをテスト対象とすると仮定しています。

まず❶で、**testコマンド**の**-z**を利用してコマンドライン引数が指定されているかチェックしています。-zは空文字列ならば真となるため、この❶のif文が真となる場合は、コマンドライン引数が指定されていません。そのためif文の中では「IPアドレスを指定してください」とエラーを出力して、exit 1としてエラー終了させています。

❷で、入力されたIPアドレスをcase文で比較しています。**case文**でパターン比較する際には、*(アスタリスク)などのワイルドカードを利用できます。そのため、「192.168.2.0/24」というネットワークは、「192.168.2.*」としてマッチできます(ただしこの書き方は、「192.168.2.AA」などIPアドレスとして不正な文字列もマッチしてしまうことに注意してください)。

❸で、指定されたIPアドレスにpingコマンドを実行しています。ここでは実行回数を指定する-cオプションを利用して、1回だけICMPパケットを送っています。この**-cオプション**の使い方などは、P.146を参照してください。なお❸では、pingコマンドの終了ステータスのみが必要で途中の出力は必要ないため、標準出力と標準エラー出力を**/dev/null**へリダイレクトして捨てています。

pingコマンドを実行した結果の終了ステータスは、シェルの特殊変数$?に入っています。❹ではこの値を比較しています。終了ステータスが0(ping成功)ならば[OK]と表示し、終了ステータスが非ゼロ(ping失敗)ならば[NG]と表示しています。

case文の分岐で、❷にマッチしなかったIPアドレス、すなわちテスト対象外ネットワークのIPアドレスは、❺の分岐にやってきます。❺は、case文にて*(アスタリスク)でマッチしていますので、その前までの条件に引っかからなかったものすべてがマッチします。ここに到達するということはテスト対象ではないアドレスということになるので、テスト対象外である旨を表示して終了しています。

こうして、IPアドレスごとに分岐させてテストを行う処理を、case文でスマートに書くことができます。

注意事項

- **このサンプル例では、入力された値がIPアドレスかどうかはきちんと確認していません(例えば、AA.AA.AA.AAという文字列が入れられるかもしれません)。**

関連項目

055 デフォルトゲートウェイにpingが通るかテストする(Linux)

No. 067 ローカルのシェルスクリプトのファイルを、リモートホストでそのまま実行する

利用コマンド
cat, ssh, hostname, ping

キーワード
SSH, リモートホスト

いつ使うか
ローカルにあるシェルスクリプトを、ssh接続先にコピーせずそのまま実行したいとき

実行例

```
$ ./script-remote.sh
server04
Ping to 192.168.2.35 : [OK]
server05
Ping to 192.168.2.35 : [NG]
server06
Ping to 192.168.2.35 : [OK]
```

スクリプト1 ..... script-remote.sh

```
#!/bin/sh

username="user1"
script="check.sh"

cat $script | ssh ${username}@192.168.2.4 "sh"
cat $script | ssh ${username}@192.168.2.5 "sh"   ❶
cat $script | ssh ${username}@192.168.2.6 "sh"
```

スクリプト2 ..... check.sh

```
#!/bin/sh

# 疎通を確認する対象サーバ
checkserver="192.168.2.35"

# スクリプトを実行するホスト名を表示する
hostname ❷

# サーバへの疎通をpingコマンドで確認する
ping -c 1 "$checkserver" > /dev/null 2>&1 ❸
```

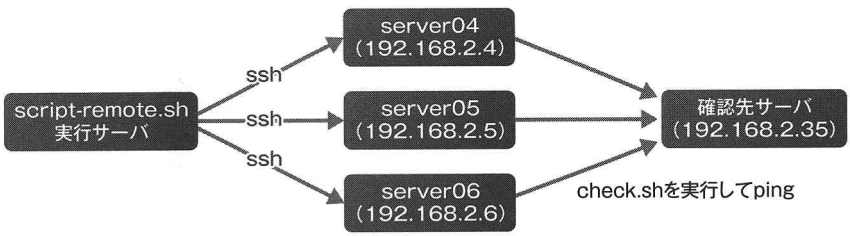
```
if [ $? -eq 0 ]; then
  echo "Ping to $checkserver : [OK]
else
  echo "Ping to $checkserver : [NG]"
fi
```

❹

解説

このスクリプトは、複数のサーバに順次ログインして、特定の宛先(スクリプト2のシェル変数checkserverで定義されている、192.168.2.35)への通信が正常に行えるかどうかをチェックするものです。

◎複数のサーバから特定の宛先に通信できるか調べる



スクリプト1を実行することで、ローカルにある確認用スクリプトファイル(ここではcheck.sh)を、sshログイン先で直接実行するように動作させているのがポイントです。このようなスクリプトの使い方は、以下のようなケースで応用できるでしょう。

- 複数のサーバで、設定内容が違っているものがないかを確認するコマンドを実行する
- ネットワーク設定が全台同じになっているか、設定ファイルを確認してチェックを行う

❶が、ローカルに設置しているシェルスクリプトのファイルを、ssh接続先で直接実行している部分です。ここでは、3台のサーバserver04(192.168.2.4)、server05(192.168.2.5)、server06(192.168.2.6)で順次実行しています。

sshコマンドは、[ユーザ名@サーバ名]の後ろに引数を付けることで、リモートサーバで実行するコマンドを指定することができます。❶では"sh"を指定して、非インタラクティブシェルを起動しています。この場合には標準入力は端末が割り当てられないため、**catコマンド**でローカルのスクリプトを標準入力へ与えてやることにより、シェルスクリプトが実行できます。

リモートサーバで実際に実行されるシェルスクリプトが、check.shです。まずはじめに、❷で**hostnameコマンド**を実行して動作しているリモートサーバのホスト名を表示しています。その後、❸で、シェル変数checkserverで指定された宛先に**pingコマンド**を

実行して疎通を確認しています。❹で、終了ステータス$?が0かどうかで成功したか失敗したかを判断し、pingコマンドが成功していれば[OK]を、失敗していれば[NG]を表示しています。このpingコマンドでの疎通確認の流れについては、P.176とほぼ同じですので、詳しくはそちらを参照してください。

このようにして、手元のシェルスクリプトのファイルを、リモート接続先にコピーせずにそのまま実行することができます。複数のサーバに同じシェルスクリプトを実行したい場合には便利でしょう。

非インタラクティブシェルとインタラクティブシェル

本項で、「リモートサーバ上でshを非インタラクティブシェルとして起動しています」と解説しました。この用語について少し補足しておきましょう。

非インタラクティブシェルとは、その名のとおりインタラクティブ(対話的)ではないシェルです。これは入力のためのキーボードや表示のための端末が接続されていないシェルで、最も身近な例はシェルスクリプトを実行するために起動されるシェルでしょう。端末から「./script.sh」などのコマンドを実行すると、そこで非インタラクティブシェルが起動されてシェルスクリプトを実行しているわけです。

一方、普段読者がキーボードを叩いて操作しているシェルは、インタラクティブシェルです。このシェルは入出力にキーボード・端末が接続されており、シェルに実行させる処理をキーボードから直接打ち込みます。

非インタラクティブシェルとインタラクティブシェルでは、起動時に読み込む設定ファイルに若干の差異があります。これらの動作の差異については、shやbashのマニュアルに記載されています。

関連項目

066 IPアドレスによる処理分岐をcase文で書く

CHAPTER

06

テキスト処理

シェルスクリプトがもっともよく利用されるのは、sedやawk、grepコマンドを組み合わせて、文字列を加工・抽出するテキスト処理でしょう。定型のレポート出力などは、スクリプトにしておけばすぐに自動化できます。この章ではCSVファイルを対象とした様々な抽出例や、ログファイルからアクセス数のレポート集計をおこなうサンプルを紹介します。

No. 068 IDが書かれたリストファイルからID抽出をする際、IDの末尾文字（下1桁）でソートして取り出す

利用コマンド
rev, sort

キーワード
末尾, ソート, IDリスト

いつ使うか
IDリストファイルの、末尾文字でソートした順に処理を行いたいとき

実行例

```
$ cat id.lst
PPX0_2
AN39_9
UIA5_3
BA06_7
QXD3_0
$ ./revsort.sh id.lst
QXD3_0
PPX0_2
UIA5_3
BA06_7
AN39_9
```

順番がばらばらのIDリストファイル

末尾の数字（下1桁）でソートされている

スクリプト

```
#!/bin/sh

# 一時ファイルの指定
tmpfile="sort.lst"

# 対象IDファイルを確認する
if [ ! -f "$1" ]; then
  echo "IDリストファイルを指定してください" >&2
  exit 1;
fi

# IDの末尾の数字でリストをソートする
rev "$1" | sort | rev > $tmpfile

# 末尾ソートされたIDリストでレポートを作成する
./report.sh $tmpfile

# 一時ファイルを削除する
rm -f $tmpfile
```

❶
❷
❸

解説

このスクリプトは、指定されたIDリストファイル（ここではid.lst）を読み込み、IDの末尾文字でソートしたリストを作成して順に処理を実行するものです。まず下1桁が"1"の人を処理し、次に下1桁が"2"の人を処理して……というように、下1桁の値で順番にリスト処理を行いたいケースを想定しています。

❸のreport.shというのは、IDリストのファイルを対象としてさまざまな処理を行うスクリプトと仮定しています。このサンプル例では、単純にファイルの中身を表示するだけのリスト1のようなスクリプトを用いました。読者の利用シーンにあわせて、この中身を実際の処理に変更してみてください。

リスト1 実行するスクリプトの例（report.sh）

```
#!/bin/sh

# 引数で指定されたファイルの中身を表示
cat "$1"
```

ユーザ管理システムの運用現場では、一部のユーザだけを選択しての処理や、アンケート集計処理などで、ユーザIDの末尾の数値（下1桁）の順に処理を行いたいという用途がときおりあります。このようなケースでは、サンプル例のように、IDリストファイルを受け取って集計を行うシェルスクリプト（ここではreport.sh）をまず作っておき、そこに用途に応じたソートをかけたIDリストファイルを渡してやる手法を用いるとよいでしょう。

このサンプル例では、まず❶で対象のIDファイルをチェックしています。**testコマンドの-fオプション**でファイルが存在するかを調べ、それを否定演算子!と併用することで、ファイルが存在しない場合にはエラーを表示して終了するようにしています。これらファイルテストの演算子については、P.110で紹介していますのでそちらを参照してください。

❷では、IDリストを末尾文字でソートするために**revコマンド**を用いています。revコマンドは単体で用いると、次のようにファイルの各行の文字列を逆転（reverse）して表示します。

revコマンドで文字列を逆転する

```
$ cat tmp.txt
abcdefghijklmnopqrstuvwxyz
1234567890
$ rev tmp.txt
zyxwvutsrqponmlkjihgfedcba
0987654321
```

使いどころが少ないコマンドですが、サンプル例のように末尾文字でソートするためには便利です。❷では、まずrevコマンドで文字列を逆転させ、パイプでつないで**sortコマ**

ンドでソートします。sortコマンドは先頭の文字でソートしますので、結果として末尾文字でソートできます。このままでは文字列が逆転したままですから、パイプ処理でもう一度revコマンドを使うことで元の文字列に戻すことができます。この出力結果を一時ファイルとして、シェル変数tmpfileというファイル名で出力して、末尾文字でソートしたIDリストファイルが得られます。

こうして末尾文字でソートしたIDリストファイルができたら、後はリストファイルを受け取って処理するスクリプト（ここではreport.sh）に渡せば、「末尾文字でソートされたID順」に処理を行うことができるわけです（❸）。

その他の末尾ソート

データベースにおいても、さまざまなIDソートの操作はよく使われます。ここではシェルスクリプトでの集計処理などで連携することが多い、MySQLでの末尾ソートの例を参考までに見てみましょう。

MySQLでは、以下のようにsubstring関数で文字列の一部を取り出すことができます。

書　式　substing関数の起動書式

```
substring(対象文字列, オフセット, 文字長)
```

この際、オフセットに負の値を指定することで、前からではなく後ろから数えた位置を指定できます。すなわち、オフセットに-1、文字長に1を指定すれば、末尾文字だけを取り出すことができます。

例えばusernfoテーブルのidカラムを末尾ソートするSELECT文は、以下のように書くことができます。

```
SELECT id FROM userinfo ORDER BY substring(id, -1, 1);
```

関連項目

042　処理開始前に、実行権限をチェックして正常動作できることを確認してから実行する

No. 069 テキストファイルから区切り文字を指定してカラムを取り出す

利用コマンド
env, grep, cut

キーワード
環境変数, 区切り文字, カラム

いつ使うか
設定されている環境変数のリストを取得し、ある変数名が定義されているかどうかを確認したいとき

実行例

```
$ ./env.sh ―――― 環境変数TMPVARが設定されていないと、start.shが実行されない
TMPVAR is null
$ export TMPVAR=1 ―― 環境変数TMPVARを設定
$ ./env.sh
TMPVAR is 1
[START] start.sh
```

スクリプト

```
#!/bin/sh

# 事前設定をしておかないとエラーになる環境変数の定義
envname="TMPVAR" ―――――――――――――――――――――――――― ❶

# env コマンドで環境変数一覧を表示し、cut コマンドで、
#   * 1番目の値を表示 [-f 1]
#   * 区切り記号は : [-d "="]
# として表示する
env | cut -f 1 -d "=" > env.lst ―――――――――――――――― ❷

# チェックする環境変数名が env.lst にマッチするかで
# 未定義かどうかを確認する
grep -q "^${envname}$" env.lst ―――――――――――――――― ❸

if [ $? -eq 0 ]; then ―――――――――――――――――――――― ❹
  # 環境変数が設定されていれば start.sh を実行
  echo "環境変数 $envname は設定されています" ―――― ❺
  ./start.sh
else
  echo "環境変数 $envname が設定されていません"
fi
```

解説

このスクリプトは、現在設定されている環境変数の一覧リストを**envコマンド**で取得し、指定した環境変数が設定されているかどうかを判断するものです。シェルスクリプトでは環境変数に頼った記述をすることが多く、その際の事前チェックなどに利用されるケースを想定しています。このサンプル例では、start.shというスクリプトを内部で実行しており、このstart.shの実行には環境変数TMPVARの設定が必要という仮定をしています。

まず❶で、事前に設定しておくべき環境変数の名前をシェル変数envnameに代入しています。ここで指定しているTMPVARという環境変数がセットされていない場合は、チェックでエラーとする仕様とします。

❷で、envコマンドを利用して設定されている環境変数を一時ファイルenv.lstに出力しています。envコマンドの出力例を以下に示します。

envコマンドで環境変数を出力する

```
$ env
HOSTNAME=www.example.com
TERM=xterm
SHELL=/bin/bash
HISTSIZE=1000
...
LOGNAME=user1
G_BROKEN_FILENAMES=1
_=/bin/env...
```

見てのとおり、envコマンドの出力結果は「環境変数名＝値」のように、＝（イコール）記号区切りとなっています。環境変数名は第1カラムにあるわけですが、このようなテキストファイルから指定したカラムの値を取り出したいときは、「区切り文字を指定して」「第n番目の値を取り出す」というコマンドで対応できます。これにはいくつか手法がありますが、このサンプル例ではcutコマンドを用いています。

cutコマンドは、その名のとおり、テキストファイルからある一部分を切り出すコマンドです。**-dオプション**で区切り文字を指定し、**-fオプション**とそれに続く数字で取り出す場所を指定します。❷では、「-d "="」と指定することで、＝（イコール）を区切りとし、「-f 1」で第1番目の値、すなわち設定されている環境変数名を抽出できるわけです。

❸で、チェックしたい環境変数名が定義済みかどうかを調べています。**grepコマンド**で、先ほど出力した環境変数を記録した一時ファイルとマッチさせることで、シェル変数envnameに定義されている値が環境変数として設定されているかどうかを判別します。ここではコマンドの終了ステータスのみを利用するため、grepコマンドは結果を表示しない**-qオプション**を利用しています。

❹でgrepコマンドの終了ステータス$?を利用して、環境変数が設定済みか未設定かで処理を分岐しています。grepコマンドで**マッチ行があれば終了ステータスは0**となってい

ますから、これはすなわちシェル変数envnameで定義した環境変数が設定済みということです。そのため❺で環境変数は設定済みと表示し、続くスクリプトstart.shを実行しています。ここではこのスクリプトは、単純に「[START] start.sh」と表示するだけのものを仮定しています。読者の環境にあわせて、このstart.shを実際に利用するスクリプトに置き換えてみてください。

注意事項

- 環境変数が設定されているかどうかは、一般的には次のように、testコマンドの空文字列かどうかを確認する-z演算子を用いて、環境変数の値が空文字列かどうかで調べる手法が一般的です。

```
if [ -z "$TESTVAR" ]; then
  echo "環境変数 TESTVAR は空"
else
  echo "環境変数 TESTVAR は設定済"
fi
```

この手法では、「環境変数が未定義」か「環境変数に空文字列が設定されている」かのどちらなのかは判別できません。ただし、実用上はそれで困ることはほぼないでしょう。

- =（イコール）区切りのテキストファイルから1つ目のカラムを取り出すには、awkコマンドの区切り文字を指定する**-Fオプション**を利用して次のようにも書けます。

```
$ awk -F= '{print $1}'
```

しかしこのサンプル例では、コンパクトでわかりやすいcutコマンドを用いる例を紹介しました。

関連項目

072 CSVファイルから、指定した特定レコードのカラムの値を得る

No. 070 ファイル先頭のシバン(#!/bin/shなど)を抽出し、スクリプトに応じた拡張子を付加する

利用コマンド
head, mv

キーワード
シバン, 拡張子, 先頭行

いつ使うか
拡張子のないスクリプトファイルに、自動的に拡張子を付加したいとき

実行例

```
$ ./shebang.sh script
`script' -> `script.sh'
$ ./shebang.sh sample1
`sample1' -> `sample1.pl'
```

スクリプト

```
#!/bin/sh

# 対象スクリプトファイルの存在を確認する
if [ ! -f "$1" ]; then ―――――――――――――――――――――――――❶
  echo "指定されたファイルが見つかりません: $1" >&2
  exit 1
fi

# ファイルの先頭行を読み出す
headline=$(head -n 1 "$1") ―――――――――――――――――――――――❷

# ファイルの先頭行ごとに拡張子を判定して付加する
case "$headline" in
  */bin/sh|*bash*)
    mv -v "$1" "${1}.sh"
    ;;
  *perl*)
    mv -v "$1" "${1}.pl"
    ;;
  *ruby*)
    mv -v "$1" "${1}.rb"
    ;;
  *)
    echo "Unknown Type: $1"
esac
```

❸

解説

このスクリプトは、スクリプトファイルの先頭行を読み込んで、使われている言語に応じた拡張子をファイル名に付けるものです。

慣例的にスクリプトファイルは、その言語に応じた拡張子をファイル名に用います。一般的によく利用される拡張子を次の表にまとめました。

広く使われている拡張子

拡張子	言語
sh	sh、bash
pl	Perl
rb	Ruby
py	Python
php	PHP

しかし拡張子はあくまで慣例であって、シェルスクリプトを含めて、スクリプトファイルは特に拡張子を付けずとも動作します。そのためシェルスクリプトを書く際、いちいち.shという拡張子を付けない人もいます。

読者が管理するシステムには、拡張子を付けないスクリプトファイルが数多くあるかもしれません。そのようなスクリプトを、ファイル名だけ見て何の言語で書かれているかわかりやすいように、一括して拡張子を付加したい場合があるかもしれません。このサンプル例はそのようなケースで役に立つでしょう。

さて、シェルスクリプトでは、はじめの1行目は必ず次のような#!で始まる行を書きます。

```
#!/bin/sh
```

これはShenbang(**シバン**あるいはシェバン)と呼ばれる形式です。UNIXではファイルを実行する際に、そのファイルがマシン語で書かれたファイルであればそのまま実行します。そうでなければ、ファイルの先頭を読み込み、それが#!であった場合はその後ろのコマンドを実行します。上記では/bin/shを実行しますから、シェルスクリプトとして動作します。

このシバンの動きを理解するために、ちょっと細かい話になりますが、以下に少し実験をしてみましょう。

C言語などで書かれた実行ファイル(マシン語のファイル)と、シェルスクリプトのファイルは、実行させるための最低限のパーミッションが異なります。C言語などで書かれた実行ファイルは、読み込みのパーミッションが付いていなくても、実行権限さえあれば実行できます。

C言語で書いた実行ファイルは実行ビットのみでも実行できる

```
$ chmod 100 a.out
$ ls -l a.out
---x------ 1 user1 user1 6425 Jan  3 20:12 a.out*
$ ./a.out
Hello, World.
```

一方、シェルスクリプトのファイルは、一般ユーザで実行する場合には実行権限だけでは実行できず、必ず読み込みの権限も必要です。

シェルスクリプトは実行ビットだけでは実行時にエラーとなる

```
$ chmod 100 ptest.sh
$ ls -lF ptest.sh
---x------ 1 user1 user1 29 Jan  3 20:14 ptest.sh*
$ ./ptest.sh
/bin/sh: ./ptest.sh: Permission denied
```

これは、UNIXではシェルスクリプトのファイルを実行する際には、まず先頭のシバンが解釈され、そこで指定されているコマンドにファイル自身を読み込ませる形で実行されるからです。つまり上記の例ならば、「$ /bin/sh ./ptest.sh」と解釈されるため、ptest.shには読み込み権限がないといけません(ただし、rootユーザならばそのまま実行できます)。

サンプル例ではこのシバンを見て、ファイルの種類を判別して拡張子を付加する処理を行っています。

まず❶で対象のスクリプトファイルをチェックしています。**testコマンド**の**-f演算子**でファイルが存在するかを調べ、それを否定演算子!と併用することで、ファイルが存在しない場合にはエラーを表示して終了するようにしています。これらファイルテストの演算子については、P.110で紹介していますのでそちらを参照してください。

続いて❷で、**headコマンド**を用いてファイルの先頭1行を取り出してシェル変数headlineに格納しています。headコマンドはファイルの先頭を読み出すコマンドで、**-nオプション**を利用することで指定した行数だけを取り出すことができます。ここでは-n 1とすることで先頭1行だけを取り出して、シバンの行だけを取得しています。

❸で、**case文**を用いてファイル種別を判断しています。「*/bin/sh|*bash*」にマッチする場合(すなわちshもしくはbashスクリプト)は拡張子を.shに、「*perl*」にマッチする場合は拡張子を.plに、「*ruby*」にマッチする場合は拡張子を.rbに、**mvコマンド**を利用して変更しています。コマンド名の後ろにも*を置いているのは、オプション指定などのために後ろにスペースやオプションが続いてもマッチするようにするためです。

なお、拡張子変更のmvコマンドには**-vオプション**を付けて、変更前のファイル名と変更後のファイル名を表示するようにしています。

❸のcase文の最後に、「*」でどれにもマッチしなかったファイルの処理として、

"Unknown Type:"と表示するだけにして、ファイル名は変更していません。このようにしてスクリプトファイルの種別を判断して拡張子を変更することができます。

注意事項

- このスクリプトは、Perl,Ruby,sh,bashにしか対応していません。また、例えばRubyが/usr/bin/perl/rubyというおかしなパスにインストールされている場合には、誤動作を起こします。
- このスクリプトは、すでに拡張子があるかどうかはチェックしていないため、拡張子があるファイルに対して実行すると「ptest.pl.pl」のように拡張子が2重になってしまいます。
- 正体不明のファイル種別の判断には、fileコマンドが便利です。fileコマンドは次のように、任意のファイルを引数に取り、そのファイルが何であるかを判断して表示してくれます。

fileコマンドの実行例

```
$ file /usr/bin/startx
/usr/bin/startx: POSIX shell script text executable

$ file network.dat
network.dat: tcpdump capture file (little-endian) - version 2.4
(Ethernet, capture length 65535)
```

上の例では、1つ目のstartxファイルはシェルスクリプト、2つ目のnetwork.datファイルはtcpdumpしたパケットキャプチャファイルであることがわかります。なおfileコマンドは、マジックファイルといういわば「ファイル辞典」を持っており、これを元にファイル種別を判断しています。興味のある方は、man magicとしてマジックファイルのマニュアルを読んでみてください。

- headコマンドの逆として、ファイルの末尾行のみを表示するtailコマンドもあります。

関連項目

042 処理開始前に、実行権限をチェックして正常動作できることを確認してから実行する

No. 071 入力ファイルのハッシュ値を、行ごとに追加カラムとして出力する

利用コマンド

paste, md5sum, read, awk

キーワード

ハッシュ , ペースト, カラム

いつ使うか

ファイルから入力データを読み込み、一行ごとにそのハッシュ値を計算してCSVファイルとして出力したいとき

実行例

```
$ cat data.txt
abcdefg
password
123456
$ ./paste.sh data.txt
abcdefg,7ac66c0f148de9519b8bd264312c4d64
password,5f4dcc3b5aa765d61d8327deb882cf99
123456,e10adc3949ba59abbe56e057f20f883e
```

スクリプト

```
#!/bin/sh

# ハッシュ値を出力する一時ファイルを初期化する
tmpfile="hash.txt"
: > $tmpfile                                    ❶

# シェルの区切り文字を改行のみとする
IFS='                                           ❷
'

# 指定されたテキストファイルから1行ずつ読み込む
while read -r line                              ❸
do
  # 各行のMD5ハッシュを取得する。
  # コマンドの後ろにはファイル名が付くため、1カラム目を取り出す
  echo -n "$line" | md5sum | awk '{print $1}' >> $tmpfile    ❺
done < $1                                       ❹

# 元のテキストファイルと、ハッシュ値を出力した一時ファイルを、
# カンマ区切りで連結して表示する
paste -d, "$1" $tmpfile                         ❻
```

解説

このスクリプトは、指定されたテキストファイルの行ごとにMD5のハッシュ値を計算し、そのハッシュ値をカンマ区切りのCSV形式で出力するものです。

MD5とはハッシュ関数の1つで、与えられた入力に対して128ビットのハッシュ値を出力します。**ハッシュ値**とはメッセージダイジェストとも呼ばれ、入力値に対してハッシュ関数による演算を行って出力される値です。メッセージが壊れていたり、改ざんされていないかを手軽にチェックできるため、広く使われています。

わずか1文字の違いでもハッシュ値は大きく変わる

```
$ echo -n "ABCDEFGABCDEFGABCDEFGABCDEFGABCDEFG" | md5sum
bd3b9bbc014d8f8ebc284d5d590bdb1a  -
$ echo -n "AACDEFGABCDEFGABCDEFGABCDEFGABCDEFG" | md5sum
70709a03675c0677ca0a1ce9aea53f75  -
```

上記が、MD5のハッシュ値を取得する**md5sumコマンド**の出力例です。ここでは"ABCDEFG"が5回繰り返される文字列のハッシュ値をまず求め、続いてこの文字列がどこかで壊れてしまったという想定で、2文字めのBがAに化けている文字列のハッシュ値を求めています。**echoコマンド**には改行を付けない**-nオプション**を付けて、文字列のみのハッシュ値が得られるようにしています。入力が1文字違うだけなのに、出力されているハッシュ値は大きく異なることがわかります。このようにハッシュ値を比べることで、メッセージが改ざんされていないか、壊れていないかを判定することができます。また出力されたハッシュ値から、逆に元の入力値を求めることは一般的に困難です。

サンプルスクリプトでは、まず入力ファイルの各行を読み込んで、各行のハッシュ値を計算して別ファイルに出力します。そのための一時ファイルの初期化を❶で行っています。**:（ヌルコマンド）**による初期化については、P.71を参照してください。

❷では、**IFS**に改行を代入することで、シェルの区切り文字を改行のみに設定しています。シェルはスペース記号をデフォルトの区切り文字とするため、このスクリプトでファイルから中身を読み込む際、単語の先頭などにスペースが入っていた場合に区切り文字とみなされてしまって正しいハッシュ値を得ることができなくなります。そのためここで区切り文字を改行のみとする設定を行っています。このIFSの設定手法については、サンプル048で説明していますのでそちらを参照してください。

❸で、while文を用いて指定された入力ファイルから1行ずつ、シェル変数lineにreadコマンドを用いて読み込みます。ここでは❹でwhile文全体に入力リダイレクトしており、コマンドライン引数に指定されたファイルから読み込むよう処理しています。なお❸の**readコマンドには**、バックスラッシュ（¥）の付いた文字をそのまま扱うように**-rオプション**を付けています。ここで-rオプションを付けないと、例えば"abcd¥nefgh"という文字列中の¥nが、'¥'と'n'という2つの文字としてではなく、'改行'として扱われてしまいます。

❺が、ハッシュ値を計算している処理部分です。md5sumコマンドに、パイプでつない

だechoコマンドを用いて入力値を与えています。なお、md5sumコマンドの出力は後ろにファイル名(この例では標準入力なので-(ハイフン))が付いています。このファイル名部分は不要なため、**awkコマンド**で1カラム目のみを取り出しています。この結果を一時ファイル$tmpfileに出力しています。

ここまでの結果で、一時ファイル$tmpfileの中身は次のように各行のハッシュ値が並んでいます。

```
7ac66c0f148de9519b8bd264312c4d64
5f4dcc3b5aa765d61d8327deb882cf99
e10adc3949ba59abbe56e057f20f883e
```

最後に❻で、元のファイルdata.txtと、ハッシュ値を記録した一時ファイル$tmpfileを連結しています。これには**pasteコマンド**を用います。pasteコマンドは、2つのテキストファイルを「横方向に連結する」コマンドです。デフォルトではタブ区切りで連結されてしまうため、ここではカンマ区切りのCSVファイルとするために区切りを指定する**-dオプション**を用いて「-d,」としています。これで、実行例に見るように、元の値とハッシュ値のCSVファイルを作ることができます。

ハッシュ値を他のスクリプトで再利用したい場合は、このサンプル例のように、いったん別ファイル(hash.txt)に出力しておきたいケースがあるでしょう。そのためこのサンプル例では、ハッシュ値を別ファイルに出力しておき、最後に元ファイルとpasteコマンドで結合する方法で処理してみました。

注意事項

- FreeBSDおよびMacでは、Linux環境とはコマンドの名前が違うため、次のようにmd5sumコマンドではなくmd5コマンドを利用してください。

```
echo $line | md5 | awk '{print $1}' >> $tmpfile
```

- MD5はハッシュ値が128ビットしかないため、安全性の問題から、現在はより出力の長いSHA形式に移行が進んでいます。しかしSHA形式を扱うコマンドは一部のOSでは標準でインストールされていないこともあり、ここではMD5を採用しました。
- ❺のようにecho -nをMacで利用する際は、No.05の注意事項を参照してください。

関連項目

027 ファイルの中身を消去して、ゼロバイトの空ファイルにする

No. 072 CSVファイルから、指定した特定レコードのカラムの値を得る

利用コマンド
cut, read, echo

キーワード
CSV, レコード, カラム

いつ使うか
CSVファイルから、引数で指定したIDに対応する特定のカラムを表示したいとき

実行例

```
$ cat data.csv
0001,Osaka,45
0002,Kyobashi,312
0003,Tenma,102
0004,Morinomiya,3
0005,Tamatsukuri,92
$ ./csv-select.sh 0004 ―― CSVファイルから、指定したIDの名前カラムを表示する
Morinomiya
```

スクリプト

```
#!/bin/sh

# CSV ファイルを指定
csvfile="data.csv"

# ID が指定されていなければ終了
if [ -z "$1" ]; then
  echo "ID を指定してください" >&2
  exit 1
fi

# CSV ファイルが存在しなければ終了
if [ ! -f "$csvfile" ]; then
  echo "CSV ファイルが存在しません: $csvfile" >&2
  exit 1
fi

while read line
do
  # 行内の各カラムを cut コマンドで取り出す
  id=$(echo $line | cut -f 1 -d ',')
  name=$(echo $line | cut -f 2 -d ',')
  score=$(echo $line | cut -f 3 -d ',')
```

❶ ❷ ❸ ❹

```
  # IDカラムが、コマンドライン引数で指定されたIDと一致する
  # 場合には、名前フィールドを表示する
  if [ "$1" = "$id" ]; then
    echo "$name" ―――――❻
  fi
done < $csvfile ―――――❺
```

解説

このスクリプトは、CSVファイルから、指定されたIDに対応する名前フィールドを取り出して表示するものです。ここで対象のCSVファイルは、次のように「ID番号,名前,スコア」という形式になっているものと仮定しています。

リスト1 サンプルで使うCSVファイルの内容

```
0001,Osaka,45
0002,Kyobashi,312
0003,Tenma,102
0004,Morinomiya,3
0005,Tamatsukuri,92
```

CSVファイルはWindows上でもMicrosoft Excelなどでよく用いられますが、簡便なデータ構造のため、UNIXでもちょっとしたレポート処理などによく利用されます。シェルスクリプトでCSVファイルを扱うには、awkコマンドを使う、IFSに,(カンマ)を設定する、などいくつかの手法がありますが、ここではcutコマンドを用いて各項目を取り出す例を見てみましょう。

まず❶で、コマンドライン引数をチェックしています。このサンプル例では検索するIDを引数で指定するので、**testコマンド**の**-z演算子**を用いて引数を確認し、空の場合はID指定するようエラーを表示して終了しています。

❷では、対象のCSVファイルの存在をチェックしています。-fは対象が通常ファイルかどうかをチェックする演算子です。それを否定演算子!と併用することで、対象がディレクトリであったり、ファイルが存在しない場合にはエラーを表示して終了するようにしています。これらファイルテストの演算子については、P.110で紹介していますのでそちらを参照してください。

続いて❸で、シェル変数lineに**readコマンド**を用いてCSVファイルを読み込みます。ここでは❺のように、while文に対して入力リダイレクトを書いています。これにより、CSVファイルからreadコマンドを用いて、シェル変数lineに1行ずつ順に読み込むことができます。

❹で、CSVファイルの1行から各項目を取り出しています。**cutコマンド**はテキストを切り出すことのできるコマンドで、**-fオプション**で取り出すフィールド番号を指定し、-d

で区切り文字を指定できます。つまり「cut -f 1 -d ','」と書くと、「カンマ区切りで1番目のフィールドの取り出し」を意味します。cutコマンドのこのような使い方は、P.185で解説していますのでそちらもあわせて参照してください。

このサンプル例では、CSVファイルは「ID番号,名前,スコア」という値が入っていると想定しましたので、❹ではこの項目それぞれをシェル変数id,name,scoreにコマンド置換$()を用いて代入しています。

こうしてCSVファイルの各項目が取り出せましたので、❻で、シェルスクリプト起動時に指定されたIDと、現在読み込んでいるCSVファイルのIDが一致するかをif文で判断しています。これが一致すれば、それが該当のIDの行ですから、先ほど代入したシェル変数nameの値をechoコマンドで表示しています。全行に対してwhile文で繰り返すことにより、CSVファイルから該当IDの名前を抽出することができます。

注意事項

- このスクリプトはマッチする行をそのまま表示するため、同一のIDを持つレコードが複数あってもそのまま複数行表示するという仕様です。
- 値自身に,(カンマ)を含むCSVファイルは、このスクリプトでは対応していません。
- このスクリプトでは、data.csvの値に複数のスペースが入っていると、それらは1つのスペースにまとめられてしまいます(例えば、"<space><space>Osaka"は、"<space>Osaka"になります)。これでは困る場合には、P.192と同様にシェルの区切り文字IFSからスペースを除外して、IFSを改行のみとしてください。
- サンプル例のように名前カラムだけを表示するのではなく、単にIDにマッチする行全体を表示したいだけならば、**awkコマンド**を使って次のように書くとよいでしょう。

```
id="$1"
awk -F, -v id="$1" '$1 == id {print}' data.csv
```

上記では、-Fオプションに,(カンマ)を設定してカンマ区切りとし、さらにawk変数を指定する-vオプションで、コマンドライン引数$1をそのままawk変数idとして扱えるようにしています。そして、$1(1カラム目)がidと一致すればprint文で表示しています。

関連項目

042 処理開始前に、実行権限をチェックして正常動作できることを確認してから実行する
069 テキストファイルから区切り文字を指定してカラムを取り出す
071 入力ファイルのハッシュ値を、行ごとに追加カラムとして出力する

No. 073 CSVファイルにIDリストを入力して、対応するIDのカラム値を得る

利用コマンド
read, grep

キーワード
IFS, CSV, レコード, カラム, 区切り文字, 分割

いつ使うか
IDリストファイルと、データCSVファイルから、該当IDの指定したカラム値を表示したいとき

実行例

```
$ cat data.csv ―――――――― データCSVファイルの確認
0001,Osaka,45
0002,Kyobashi,312
0003,Tenma,102
0004,Morinomiya,3
0005,Tamatsukuri,92
$ cat id.lst ―――――――― 入力IDファイルの確認
0003
0004
$ ./csv-list.sh id.lst ―――― IDリストに一致する名前表示
Tenma
Morinomiya
```

スクリプト

```
#!/bin/sh

filecheck()
{
  if [ ! -f "$1" ]; then
    echo "ERROR: File $1 does not exist." >&2
    exit 1;
  fi
}

# データCSVファイル名と、IDリストファイル名を定義して、
# ファイルの存在チェックを行う
csvfile="data.csv"
idlistfile="$1"
filecheck "$csvfile"
filecheck "$idlistfile"
```

❷
❶

```
while IFS=, read id name score ———————❸
do
  grep -xq "$id" "$idlistfile" ———————❹
  if [ $? -eq 0 ]; then ———————————————❺
    echo $name
  fi
done < "$csvfile"
```

解説

このスクリプトは、IDリストファイルをコマンドライン引数に指定することにより、**CSVファイル**から一致するIDのカラム値をまとめて取得するものです。ここで対象のCSVファイルは、次のような「ID番号,名前,スコア」という形式になっているものと仮定しています。

リスト1 CSVファイル(data.csv)

```
0001,Osaka,45
0002,Kyobashi,312
0003,Tenma,102
0004,Morinomiya,3
0005,Tamatsukuri,92
```

またIDリストファイルは次のように、抽出したいIDが書かれたテキストファイルを想定します。

リスト2 抽出したいIDのリストファイル(id.lst)

```
0003
0004
```

CSVファイルは、UNIXでもちょっとしたレポート処理などによく利用されます。このサンプル例では、**IFS**を,(カンマ)に設定して、**readコマンド**でCSVファイルを扱う方法を見てみましょう。

❶で、シェル変数csvfileにデータCSVファイル名を、シェル変数idlistfileにIDリストファイル名を設定して、ファイルの存在チェックを行っています。存在チェックはシェル関数filecheck()で処理しています。

シェル関数filecheck()では、testコマンドの-f演算子(対象が通常ファイルかどうかをチェック)を利用して対象ファイルの確認をしています(❷)。否定演算子!と併用することで、対象がディレクトリであったり、ファイルが存在しない場合にはエラーを表示して終

了します。これらファイルテストの演算子については、P.110で紹介していますのでそちらを参照してください。

❸でCSVファイルからシェル変数id、name、scoreに対して値を読み込んでいます。この行を理解するために、まず次のような「**一時的に環境変数を設定してコマンドを実行する**」書き方をはじめに説明します。サンプル例では、この一時的に設定する環境変数として、IFSを利用しているわけです。

一時的に環境変数を設定してコマンドを実行する例

```
$ TMPDIR=/mytmp ./start.sh
```

上記は、環境変数TMPDIRを一時的に設定してstart.shを実行します。この行より後では、環境変数TMPDIRに設定されていた値は元から設定されていた値となり、/mytmpにはなりません。つまり、「環境変数=値 コマンド」とすれば、ある特定のコマンドやスクリプトを実行するときだけ環境変数を一時的に設定できるわけです。

❸に戻ると、ここではまずwhile文を実行して、ループの終わりのdoneのところで$csvfileを入力リダイレクトしています。これにより、while文で、シェル変数csvfileで指定されたファイルを1行ずつ読み込んで実行することができます。

この❸のwhile文の条件式は、次のようになっています。

```
IFS=, read id name score
```

すなわち、環境変数IFSを一時的に,(カンマ)に設定してreadコマンドを実行しています。ここで、id、name、scoreはシェル変数です。

環境変数IFSに,(カンマ)を一時的に設定することで、シェルが解釈する区切り文字を,(カンマ)だけにすることができます。これにより、カンマ区切りの行を、カンマで値を分割してそれぞれのシェル変数に代入できます。なおここで利用しているIFSについて詳しくは、P.125で解説していますので、そちらも参照してください。

以上より、❸のwhile文は日本語で詳しく書き示すと、

「シェル変数csvfileで指定されたCSVファイルから1行ずつ読み込んで、readコマンドを実行してシェル変数に値を代入する。この際、値の区切りは、環境変数IFSに,(カンマ)を設定してカンマ区切りとする。これによりカンマ区切りの行が分割され、それぞれのカラム値がシェル変数id、name、scoreに代入される」

という意味になります。少々複雑ですが、❸のような記法はCSVファイルを扱う際にはよく使われる書き方ですので、じっくり読んで理解してみてください。

❹では、IDリストファイルのIDと、CSVファイルのIDをマッチさせるために、**grepコマンド**の-xオプションと-qオプションを利用しています。grepコマンドの**-xオプション**は、行全体がパターンと完全一致する場合のみを選択するオプションです。grepでは、例え

ば"0001"というIDで検索すると、"00010"というIDにもマッチしてしまうため、ファイルからID検索する際には誤った結果を出力してしまう場合があります。CSVファイルから抽出したIDが、IDリストファイルと完全一致するのをチェックするために-xオプションを利用しているわけです。

また❹では、マッチしたか/しないかの結果のみを終了ステータスで利用するため、grepコマンドの検索結果を出力しない**-qオプション**を用いています。

❺でgrepコマンドの終了ステータスを判断して、マッチしたかしていないかを判断しています。マッチしていれば終了ステータス**$?**が0となっていますから、これをif文で判定して真の場合にはCSVファイルから取り出した名前(シェル変数name)をechoコマンドで表示して、該当IDの名前を抽出することができるのです。

注意事項

- 値自身に,(カンマ)を含むCSVファイルは、このスクリプトでは対応していません。
- サンプル例のように名前のみを抽出して表示するのではなく、単純に入力IDファイルにマッチする行全体を表示したいときは、grepコマンドの-fオプションが利用できます。

IDリストを-fオプションで指定

```
$ grep -f id.lst data.csv
0003,Tenma,102
0004,Morinomiya,3
```

ただしサンプル例のケースでは、id.lstファイルに例えば[0001]と書いてあると、[00010]というIDもマッチしてしまいます。また、もしも名前に [0001Tenma]というものがあった場合にもマッチしてしまいます。このようなケースへの対策としては、IDリストファイルは先頭に^を、ID末尾にカンマを付けて[^0001,]と書いておく、などの手法があります。

関連項目

042 処理開始前に、実行権限をチェックして正常動作できることを確認してから実行する
048 .svnなどの隠しファイル・ディレクトリのみを列挙する
072 CSVファイルから、指定した特定レコードのカラムの値を得る

No. 074 数値データの書かれたCSVファイルから平均値を計算する

利用コマンド
awk

キーワード
平均値, CSVファイル

いつ使うか
CSVファイルから、特定のカラム値の平均値を計算してファイルとして出力したいとき

実行例

```
$ cat data.csv                          データファイルの確認
0001,Osaka,45
0002,Kyobashi,312
0003,Tenma,102
0004,Morinomiya,3
$ ./csv-avg.sh data.csv
$ cat data.avg                          平均値が書かれた[ファイル名.avg]
115.5                                   というファイルが作成されている
```

スクリプト

```
#!/bin/sh

# CSVファイルが存在しなければ終了
if [ ! -f "$1" ]; then
  echo "対象のCSVファイルが存在しません: $1" >&2
  exit 1
fi

# 拡張子を除いたファイル名を取得
filename=${1%.*}

awk -F, '{sum += $3} END{print sum / NR}' "$1" > ${filename}.avg
```

❶ ❷ ❸

解説

このスクリプトは、コマンドライン引数で指定されたCSVファイルの第3カラムから、スコアの平均値を計算して出力するものです。平均値は、元のファイル名に拡張子[.avg]を付けた別ファイルとして出力します。大量の**CSVファイル**ひとつひとつに平均値を出したいときなどに役立つでしょう。

ここで対象のCSVファイルは、P.196と同様に「ID番号,名前,スコア」という形式になっているものと仮定しています。

シェルスクリプトで数値計算を行うには、exprコマンドがよく使われます。しかしexprは整数計算のみに対応しており、小数が含まれる計算には利用できません。シェルスクリプトで小数を含む計算をするには、高機能な数値計算が行えるbcコマンドを利用するケースもありますが、ここでは手軽なawkコマンドを利用してみましょう。

❶で、対象のCSVファイルの存在をチェックしています。-fは対象が通常ファイルかどうかをチェックする演算子です。それを否定演算子!と併用することで、対象がディレクトリであったり、ファイルが存在しない場合にはエラーを表示して終了するようにしています。これらファイルテストの演算子については、P.110を参照してください。

❷では、ファイルの拡張子を除いたファイル名を取得しています。これはシェルのパラメータ展開を利用した文字列置換で、シェル変数$1から、「.（ドット）に続く任意の文字列」を取り除きます。すなわち$1でコマンドライン引数としてCSVファイルが渡されていますから、この拡張子（.csv）を削ることができます。ここで用いている**パラメータ展開**の記法は、P.62で説明していますので、詳しくはそちらを参照してください。

❸が、平均値を計算している**awkコマンド**の行です。まずawkコマンドで区切り文字を,（カンマ）とするために、**-F**の区切り指定オプションで,を設定しています。awkコマンドの記法では、**{}は各行で実行され、END{}は最終行を読んだ後に実行**されます。まずは{sum += $3}として各行で、$3（3カラム目、ここではスコアの値）を変数sumに足し上げています。

ENDで用いている**NR**とは、awkの組み込み変数で、現在処理した行番号が入っています。すなわち「END{print sum / NR}」が実行されるのは最終行のため、NRはファイルの行数となります。このNRの値でsumを割ることで、スコアの平均値を表示しています。

スコアの平均値は、先ほど拡張子を除いたファイル名に拡張子.avgを付けてリダイレクトして、ファイルを出力しています。

注意事項

- awkで区切りを,（カンマ）にする際には、「-F,」と書かずに「-F ','」と指定してもかまいません。どちらの書き方でもよいのですが、他人の書いたスクリプトを読めるように、2種類の指定の仕方があることは覚えておきましょう。
- 値自身に,（カンマ）を含むCSVファイルは、このスクリプトでは対応していません。
- このサンプル例では、CSVファイルの中身のチェックはしていません。もしCSVファイルの中身が空っぽの場合は、0÷0を実行することになりエラーとなります。

関連項目

042 処理開始前に、実行権限をチェックして正常動作できることを確認してから実行する
072 CSVファイルから、指定した特定レコードのカラムの値を得る

No. 075 数値データ（CSVファイル）から、"*"を利用して簡単なテキストグラフを出力する

利用コマンド
echo, awk, sort, head, expr, read

キーワード
グラフ, CSVファイル, 最大値

いつ使うか
数値データを、端末上で簡単なテキストグラフとして表示したいとき

実行例

```
$ ./csv-graph.sh data.csv
******* [Osaka]
************************************************** [Kyobashi]
*************** [Tenma]
 [Morinomiya]
************** [Tamatsukuri]
```

スクリプト

```
#!/bin/sh

csvfile="data.csv"    # データCSVファイル
GRAPH_WIDTH=50        # グラフの横幅値

markprint() {
  local i=0
  while [ $i -lt $1 ]
  do
    echo -n "*"
    i=$(expr $i + 1)
  done
}

# データから最大値を取得する。逆順ソートして先頭を取ればよい
max=$(awk -F, '{print $3}' "$csvfile" | sort -nr | head -n 1)

# データのすべてが0の場合は最大値を1とする
if [ $max -eq 0 ]; then
  max=1
fi

# CSVファイルを読み込み、値ごとの値をグラフ出力する
```

❶ ❷ ❸ ❹ ❺ ❻ ❼

```
while IFS=, read id name score ―――――――――――――――――――――❽
do
  markprint $(expr $GRAPH_WIDTH ¥* $score / $max) ――――――――――❾
  echo " [$name]"
done < $csvfile
```

解説

このスクリプトは、コマンドライン引数で指定されたCSVファイルのスコア値を、"*"でテキストグラフとして出力するものです。

グラフはExcelなどで書く人が多いでしょうが、メール本文中でちょっとしたグラフを書いたり、端末上でまず様子を見たいというときには、このような昔ながらのテキストグラフもなかなか役に立ちます。スクリプトとして使えるようにしておくと便利でしょう。

ここで対象のCSVファイルは、次のような「ID番号,名前,スコア」という形式になっているものと仮定しています。

リスト1 CSVファイル（data.csv）の内容

```
0001,Osaka,45
0002,Kyobashi,312
0003,Tenma,102
0004,Morinomiya,3
0005,Tamatsukuri,92
```

❶でまず、スクリプトの初期設定を行います。シェル変数csvfileには入力するデータCSVファイル名を、シェル変数GRAPH_WIDTHにはグラフの横幅値を設定します。

❷は、テキストグラフを出力するシェル関数を定義しています。この関数は数値を1つ受け取って、その数だけ"*"を表示するだけの関数です。関数への引数は、シェルにより$1に代入されています。

❸では、まずカウンタ変数を**ローカル変数**として初期化しています。この**local宣言**の使い方は、P.33を参照してください。❹で、関数への引数$1よりカウンタが小さいあいだ、"*"を出力し続けています。なお"*"を出力する際は、**echoコマンド**の**-nオプション**を利用して改行しないようにしています（❺）。このechoコマンドの-nオプションの例は、Macではエラーとなりますので、「注意事項」を参照してください。

❻は、CSVファイルから事前に最大値を取得する処理です。テキストグラフを書く際は、横幅値を決めてその中に収まるようにしなければいけません。そのため、ここでまずデータの最大値を取得して、これが横幅に収まるようにします。

CSVの、特定のカラム列から最大値を取得するには、

1） そのカラム値を1列に表示

2） sortコマンドの-nr（数値ソートかつ逆順ソート）オプションでソートする
3） headコマンドで1行目のみを取り出す

と順番に処理すれば最大値が得られます。❻はこの処理を行っています。まず**awkコマンド**の区切りを**-F**,で,（カンマ）として、3カラム目を{print $3}として表示しています。これをパイプで**sortコマンド**に渡し、**-nrオプション**により数値で逆順ソートしています。最後に**headコマンド**の指定行数を取り出す**-nオプション**で1を指定し、先頭1行目を取り出します。これでCSVファイルのスコア値のうち、最大値をシェル変数maxに代入できました。

❼では、❻で得たデータ最大値が0の場合に、シェル変数maxに1を設定しています。グラフ横幅の計算では最大値で割り算を行うため、全データが0の場合は0の割り算となりエラーとなってしまうための対応です。

これでグラフ描画の準備ができましたので、❽でデータCSVファイルから順番にデータを読み込んでグラフを描きます。❽では**IFS**に一時的に,（カンマ）を設定して、シェル変数id,name,scoreに値を読み込みます。この書き方はP.125で詳しく解説していますのでそちらを参照してください。

❾で、テキストグラフを出力するためにmarkprint関数に"*"の出力個数を渡します。scoreの数そのままを指定すると横幅があふれてしまいますので、正規化のために最大値maxで割った値を**exprコマンド**で計算し、その値をmarkprint関数に渡します。こうして横幅をできるだけいっぱいに使ってテキストグラフを出力できます。

なお、markprint関数では"*"を表示するだけで、改行は行っていません。そのため❾の後に、CSVファイルの2カラム目（名前カラム）から取り出した値がシェル変数nameに入っていますので、これを[$name]として"*"の右側に表示しています。

こうして行ごとに必要な数だけ"*"を出力し、テキストグラフを描画することができます。

注意事項

- このスクリプトでは、3カラム目のスコア値はexprコマンドで割り算しているため、整数のみ対応しています。小数は扱えません。
- ❺で、Macで改行させないメッセージを出力するecho -nを利用する際は、P.15を参照してください。

関連項目

012 関数の中でローカル変数を定義して、呼び出し元の変数を破壊しないようにする
072 CSVファイルから、指定した特定レコードのカラムの値を得る
073 CSVファイルにIDリストを入力して、対応するIDのカラム値を得る

No. 076 ログファイルのカラム位置を入れ替えて出力し、見やすく加工する

利用コマンド
awk

キーワード
アクセスログ, ログ解析, 整形

いつ使うか
Apacheのアクセスログから必要なカラムの抜き出し・並べ替えを行いたいとき

実行例

```
$ cat access_log
xx.xx.xx.xx - - [06/Jan/2014:05:58:35 +0900] "GET / HTTP/1.1" 200 83 "-"
"-"
yy.yy.yy.yy - - [06/Jan/2014:06:01:43 +0900] "GET /index.html HTTP/1.1"
200 304
yy.yy.yy.yy - - [06/Jan/2014:06:01:44 +0900] "GET /title.gif HTTP/1.1"
200 763
$ ./log-column.sh access_log
$ cat access_log.lst
[06/Jan/2014:05:58:35 +0900] xx.xx.xx.xx
[06/Jan/2014:06:01:43 +0900] yy.yy.yy.yy
[06/Jan/2014:06:01:44 +0900] yy.yy.yy.yy
```

ログファイルから、時刻とリモートホストのみ抜き出して順番を入れ替えた

スクリプト

```
#!/bin/sh

# ログファイルが存在しなければ終了
if [ ! -f "$1" ]; then
  echo "対象のログファイルが存在しません: $1" >&2
  exit 1
fi

# リクエスト時刻とリモートホストを外部ファイルへ出力
awk '{print $4,$5,$1}' "$1" > "${1}.lst"
```

❶ ❷

解説

このスクリプトは、**Apache**のアクセスログから必要な列を抜き出し、順番を入れ変えて出力するものです。ログ解析においては、同じ条件で大量のデータファイルから抽出を行うことが多く、このようにスクリプトにしておくと一括処理できて便利でしょう。

Apacheはアクセスログをさまざまにカスタマイズできるため、この実行例で扱っているログ形式は読者の環境とは多少違っているかもしれません。ここでは、アクセスログは次のような形式となっていると仮定します。これはApacheのアクセスログとして比較的ポピュラーなcommonという設定です。

Apacheのcommon形式ログの例

```
192.168.1.1 - - [06/Jan/2014:05:58:35 +0900] "GET /index.htm HTTP/1.1" 200 83
```

このログの左から順に見た各項目の意味を、次の表に示しました。

Apacheログの読み方

意味	値の例
リモートホスト	192.168.1.1
identdによるリモートユーザ名	-
認証によるリモートユーザ名	-
リクエストを受け付けた時刻	[06/Jan/2014:05:58:35 +0900]
リクエストの最初の行	"GET /index.htm HTTP/1.1"
HTTPステータス	200
レスポンスのバイト数	83

「identdによるリモートユーザ名」は、mod_identというApacheモジュールから提供されます。現在では、Apacheでこのidentdによりユーザ名を取得するケースはほとんどありませんから、読者の環境でも単に「-」が出力されているでしょう。

「認証によるリモートユーザ名」は、BASIC認証などで入力されたユーザ名です。認証のかかっていないページでは、単に「-」が出力されます。

「リクエストの最初の行」は、「HTTPメソッド名 リクエストURI（ファイル名）HTTPバージョン」のセットで記述されます。我々がふだんWebブラウザでWebページを閲覧する際には、このリクエスト行を直接見る機会はありません。しかし内部的には、WebブラウザはWebサーバに対して、都度このようなリクエストを発行してWebページを取得しているのです。

サンプル例では、このアクセスログを元にして、1カラム目にリクエスト時刻を、2カラム目にリモートホストを、「（元のログファイル名）.lst」というファイル名で出力したいと仮定しています。読者のニーズに応じて、この形式は適宜読み替えてください。

まず、このスクリプトは引数にログファイルをとるため、❶でコマンドライン引数を確認し、ファイルの存在をチェックします。**-f**は対象が通常ファイルかどうかをチェックする演算子です。それを否定演算子!と併用することで、対象がディレクトリであったり、ファイルが存在しない場合にはエラーを表示して終了するようにしています。これらファイルテストの演算子については、P.110で紹介していますのでそちらを参照してください。

❷で、ログファイルから**awkコマンド**で必要な列を抽出しています。awkコマンドでは、アクションと呼ばれる中カッコ{}で囲った部分でさまざまな出力が行えます。ここで記述

している"$4,$5,$1"というのはawkコマンドの組み込み変数で、それぞれ第4カラム、第5カラム、第1カラムを意味します。

awkコマンドでは、空白はデフォルトの区切り文字として扱われます。このログの例では、リクエスト時刻[06/Jan/2014:05:58:35 +0900]は、時差を示す"+0900"の前にスペースがあるため、awkでは別カラムと扱われてしまいます。そのため、第4カラムと第5カラムを並べて"$4,$5"としてprintしています。なお、awkコマンドではprintする際に,(カンマ)はスペースになります。ここで、次のようにスペース記号でprintする変数を区切ると、スペースは無視されて詰められてしまうので注意してください。

awkコマンドでprintする際にスペースを使うと無視される

```
$ awk '{print $4 $5 $1}' access_log
[06/Jan/2014:05:58:35+0900]xx.xx.xx.xx
[06/Jan/2014:06:01:43+0900]yy.yy.yy.yy
[06/Jan/2014:06:01:44+0900]yy.yy.yy.yy
```

このようにして、awkコマンドで表示したいカラム位置を好きな順番で並べることで、好みのパターンにログファイルを整形することができます。

なお、❷においては**$1**という記述が、awkの変数と、シェルスクリプトの位置パラメータ変数の2つ登場しています。この2つは全く違うものですから、取り違えて混乱しないようにしてください。繰り返しになりますが、awkのprint文の中にある$1はawkの変数で、「1カラム目の値」を意味します。一方、シェルスクリプト中の$1は「1つ目のコマンドライン引数」です。

これがわかりにくければ、❷の部分は次のようにファイル名を別変数にして書いたほうがよいかもしれません。

```
logfile="$1"
awk '{print $4,$5,$1}' "$logfile" > "${logfile}.lst"
```

関連項目

042 処理開始前に、実行権限をチェックして正常動作できることを確認してから実行する
077 Webサーバのログファイルから特定のステータスコードを返しているものだけを取得する
079 Webアクセスログからファイルごとのアクセス回数を集計する

No. 077 Webサーバのログファイルから特定のステータスコードを返しているものだけを取得する

利用コマンド
awk

キーワード
アクセスログ, ログ解析, 整形

いつ使うか
Apacheのアクセスログから、ステータス404 (Not Found) のエラーを返しているリクエスト行を加工して、ファイル名のみ抽出したいとき

実行例

```
$ cat access_log
xx.xx.xx.xx - - [06/Jan/2014:05:58:35 +0900] "GET / HTTP/1.1" 200 83
yy.yy.yy.yy - - [06/Jan/2014:06:01:43 +0900] "GET /index.html HTTP/1.1"
200 304
yy.yy.yy.yy - - [06/Jan/2014:06:01:44 +0900] "GET /tittle.gif HTTP/1.1"
404 763
yy.yy.yy.yy - - [06/Jan/2014:06:01:44 +0900] "GET /title.gif HTTP/1.1"
200 763
$ ./log-select.sh
$ cat access_log.404
/tittle.gif
```

スクリプト

```
#!/bin/sh

logfile="access_log"

# ログファイルが存在しなければ終了
if [ ! -f "$logfile" ]; then
  echo "対象のログファイルが存在しません: $logfile" >&2
  exit 1
fi

# ログファイルから、HTTPステータスを外部ファイルへ出力
awk '$(NF-1)==404 {print $7}' "$logfile" > "${logfile}.404"
```

❶
❷

解説

このスクリプトは、**Apache**のアクセスログから、**HTTPステータス**が404 (Not Found) となっているファイルを抽出するものです。抽出したファイルリストは、「(元のログファ

イル名).404」というファイル名で出力します。HTTPステータス404 (Not Found)が返されているリクエストを分析することは、リンク切れを起こしているコンテンツを探すには有効な手法です。そのため定期的にこのようなスクリプトを動かしてログ監視を行う事例も多いでしょう。

このサンプル例では、Apacheのアクセスログが、次のような形式となっていると仮定します。読者の環境によって、カラム位置などは読み替えてください。

リスト1 Apacheのcommon形式ログの例

```
192.168.1.1 - - [06/Jan/2014:05:58:35 +0900] "GET /index.htm HTTP/1.1" 200 83
```

この形式では、HTTPステータス番号は後ろから2番目のカラムに入っています。そこで、後ろから2番目のカラムに"404"が出力されているログから、ファイル名を抽出してみます。なお"404"という文字列を単純にgrepすると、例えば"menu4040.html"などファイル名に404を含むファイルも抽出されてしまいます。そこで本サンプルは、特定のカラムの値が404にマッチする行のみを抽出しています。

❶ではまずログファイルの存在をチェックします。**-f**は対象が通常ファイルかどうかをチェックする演算子です。それを否定演算子!と併用することで、対象がディレクトリであったり、ファイルが存在しない場合にはエラーを表示して終了するようにしています。これらファイルテストの演算子については、P.110を参照してください。

❷がawkコマンドによるログ抽出の処理です。この行を理解するために、まず**awkコマンドの組み込み変数NF**について説明します。

NFは、awkコマンドで**現在処理している行のカラム数**(フィールド数)を指します。例えば「{print $NF}」と書くことで、最終カラムを表示できます。サンプル例では"$(NF-1)"として、このNFから1を引いた値を指定していますので、これは「後ろから数えて2カラム目」を意味します。ログ形式で見たように、ここにHTTPステータスコードが入っています。「注意事項」にも記載していますが、HTTPのリクエスト部分に入る空白文字は実にさまざまであるため、ここではカラムを前から数えずに、後ろから数えてできるだけ正確に値がとれるようにしています。

変数NFの使い方がわかったところで、❷に戻ります。awkコマンドでは、アクション{}の前に条件式を指定することができます。これにより、カラム単位でのgrep相当のことを行うことができます。ここでは「$(NF-1)==404」と条件式を書くことで、「後ろから2番目のカラムが404だったとき」、すなわち「HTTPステータスが404だったリクエスト」を指定しています。表示するカラムは、ここではファイル名のみを出力したいため、第7カラム($7)のみをprintしています。

このようにしてアクセスログから、404 Not Foundだったリクエストのファイル名のみが抽出できます。定期的にこのようなスクリプトを実行することで、リンク切れを探す手がかりとなるでしょう。

注意事項

- Webサーバには、攻撃を意図してRFC違反の妙なリクエストもときおりやってきます(GETの後ろにスペースがない、など)。この場合にはスペースの位置が違うため、このスクリプトでは適切に扱えません。
- HTTPステータス(→P.328参照)に500番台のエラーを返しているリクエストを取得するには、次のように書くとよいでしょう。

```
awk '$(NF-1)>=500 {print $7}'
```

特に500番台のエラーは、アプリケーションに問題があったり、サーバが高負荷になっているなど、何らかの異常が起きているときに出力されるエラーコードであるため運用上重要なログです。そのためこのようなスクリプトで適宜抜き出して、アプリケーション状態を監視するためにも使えるでしょう。

関連項目

042 処理開始前に、実行権限をチェックして正常動作できることを確認してから実行する
076 ログファイルのカラム位置を入れ替えて出力し、見やすく加工する
079 Webアクセスログからファイルごとのアクセス回数を集計する

No. 078 システムログからIPアドレスごとのアクセス回数を集計する

利用コマンド
sed, sort, uniq

キーワード
SSH, 不正アクセス, 認証失敗, ログ抽出

いつ使うか
sshdのログファイルから、パスワード認証に失敗しているIPアドレスをカウントしたいとき

実行例

```
# ./ssh-fail.sh
  15 10.211.55.2
   6 10.211.55.21
   2 10.211.55.18
```

接続元IPアドレスごとの、sshパスワード認証失敗回数を多い順に表示する

スクリプト

```
#!/bin/sh

# sshdのログファイル
securelog="/var/log/secure"

# IPアドレスを抜き出すためのパターンマッチ。長いため変数として格納
pattern="^.*sshd¥[.*¥].*Failed password for.* from ¥(.*¥) port .*" ―❶

# パスワード認証失敗ログからIPアドレスを抽出し、カウントして表示する
sed -n "s/$pattern/¥1/p" "$securelog" | sort | uniq -c | sort -nr ―❷
```

解説

このスクリプトは、**sshd**のログ(/var/log/secure)からパスワード認証に失敗したログを抽出し、その接続元IPアドレスをカウントして表示するものです。

インターネット上にssh接続できるサーバを設置すると、ブルートフォースアタックが毎日のようにやってくることは珍しくありません。そこで、このようなログ整形スクリプトを設置しておき、パスワード認証に失敗したアクセス元を定期的にレポート出力しておくと、不正アクセスの監視に役立つことでしょう。

sshでの接続ログは、Linux (CentOS)サーバでは一般的に**/var/log/secure**に、FreeBSDでは**/var/log/auth.log**に出力されており、次のような形式になっています。

リスト1 sshの接続ログ例

```
Jan  3 21:40:00 cent unix_chkpwd[1480]: password check failed for user (user1)
Jan  3 21:40:00 cent sshd[1478]: pam_unix(sshd:auth): authentication failure;
logname= uid=0 euid=0 tty=ssh ruser= rhost=10.211.55.2  user=user1
Jan  3 21:40:02 cent sshd[1478]: Failed password for user1 from 10.211.55.2
port 53639 ssh2
Jan  3 21:40:05 cent sshd[1479]: Connection closed by 10.211.55.2
```

"Failed password for user1 from ...."というのがパスワード認証に失敗したことを意味しますので、ここではこのアクセス元をカウントしてみましょう。なお、sshでは鍵認証などさまざまな認証方式があるため、出力されるログメッセージも環境によって変わってきます。パターンマッチさせる文字列は、読者の環境やニーズにあわせて適宜修正してみてください。

このサンプルで扱うログファイルは、「Failed password for user1 from 10.211.55.2」のようにfromの後ろにIPアドレスが入っていますので、これを取り出すことにします。パターンマッチが長い文字列となってしまうため、❶でパターンを文字列としてシェル変数に格納しています。後のsedコマンドを見やすくするため、長いパターンはこのようにシェル変数に入れておくとよいでしょう。

❷では、接続元IPアドレスを抜き出すために、**sedコマンド**でマッチ部分のみを表示する**-nオプション**と**pフラグ**の組み合わせを用いています。また同時に後方参照を用い、「from ¥(.*¥)」としてfromの後ろの部分文字列を¥1として参照して出力することで、ログファイルからIPアドレスを抜き出しています。このようにsedコマンドの-nオプションとpフラグでパターンマッチの一部分を取り出す手法は、P.46で詳しく説明していますのでそちらを参照してください。

❷のsedコマンドの出力は、パイプでsortコマンドとuniqコマンドに渡しています。まず1番目の**sortコマンド**によりIPアドレスでソートをかけ、続く**uniqコマンド**の行数をカウントする**-cオプション**を利用して、同一行の出現回数をカウントしています。

そして❷ではパイプの最後でもう一度、sortコマンドの**-nオプション**（数値ソート）と-rオプション（逆順ソート）をかけることで、アクセスが多い順に表示されるようにしています。この、「sort | uniq -c | sort -nr」は、テキスト処理で大変よく使われる慣用句のようなものですから、ぜひ覚えておきましょう。

sort -nrで、先頭の数値で降順ソートする

```
$ cat log.txt
   2 10.211.55.18
  15 10.211.55.2
   6 10.211.55.21
$ sort -nr log.txt
  15 10.211.55.2
   6 10.211.55.21
   2 10.211.55.18
```

こうして、接続元IPアドレスごとのカウント数を調べることができました。このサンプルではsshdのログを用いましたが、他にも不正アクセスの回数などを監視するため、ログファイルの抽出に応用してみてください。

sshの不正アクセスについて

本項ではsshdを取り上げましたので、この不正アクセスについて少し補足しておきましょう。

昨今はクラウドやVPSなどで手軽にサーバが手配できるようになったことから、それらサーバへの不正アクセスも急増しています。特にsshは、攻撃者が不正ログインに成功すると、容易に攻撃の踏み台として利用できてしまうため注意が必要なサービスです。

sshへのセキュリティ対策としては、

1) ポート番号をデフォルトの22/tcpから変更する
2) 公開鍵認証を利用してパスワード認証を禁止する
3) iptablesなどで接続元IPアドレスを限定する

などの手法が挙げられます。これに加えて、本項のようなログ監視も合わせて行い、不正侵入がないかどうかの確認を定期的にチェックしたほうがよいでしょう。

注意事項

- /var/log/secureは通常、rootユーザでしか読めないため、このスクリプトもroot権限で実行する必要があります。
- 同一コネクションで複数回試行されたり、短時間に何度もアクセスされた場合、ログ出力上それらは「3 more authentication failure」のようにまとめられてしまいます。そのためここでのカウント数は正確な不正アクセス試行回数ではなく、目安と考えてください。

関連項目

018 HTMLファイルから、タグの中に書かれたコマンドを抜き出してそのまま実行する
079 Webアクセスログからファイルごとのアクセス回数を集計する

No. 079 Webアクセスログからファイルごとのアクセス回数を集計する

利用コマンド
awk, sort, uniq

キーワード
アクセスログ, ログ解析, ページビュー

いつ使うか
Apacheのアクセスログからページビューを集計したいとき

実行例

```
$ cat access_log
xx.xx.xx.xx - - [06/Jan/2014:05:58:35 +0900] "GET / HTTP/1.1" 200 83 "-"
"-"
yy.yy.yy.yy - - [06/Jan/2014:06:01:43 +0900] "GET /index.html HTTP/1.1"
200 304
yy.yy.yy.yy - - [06/Jan/2014:06:01:44 +0900] "GET /title.gif HTTP/1.1"
200 763
(省略)

$ ./log-accessfile.sh
  29 /news.html
  22 /
  18 /favicon.ico
   8 /menu/
   8 /title.gif
(省略)
```

スクリプト

```
#!/bin/sh

logfile="access_log"

# ログファイルが存在しなければ終了
if [ ! -f "$logfile" ]; then
  echo "対象のログファイルが存在しません: $logfile" >&2
  exit 1
fi

# ログファイルから、GETメソッドで取得されたファイルのアクセス回数を集計する。
# awkコマンドでファイルを取り出し、sort+uniqでカウント後に降順ソートする
awk '$6=="¥"GET" {print $7}' "$logfile" | sort | uniq -c | sort -nr
```

❶ `if [ ! -f "$logfile" ]; then` から `fi` まで

❷ `awk '$6=="¥"GET" {print $7}' "$logfile" | sort | uniq -c | sort -nr`

解説

このスクリプトは、**Apache**のアクセスログから、ファイルごとのアクセス数を集計するものです。Webページを運用する際には、ページビュー（PV）を調べるのはとても重要な運用業務です。まずはこのような簡単なスクリプトで集計レポートを定期的に取得するだけでも、いろいろな情報が得られるでしょう。

このサンプル例では、Apacheのアクセスログがサンプル076と同じく、次のような形式となっていると仮定します。読者の環境によって、カラム位置などは読み替えてください。

リスト1 Apacheのcommon形式ログの例

```
192.168.1.1 - - [06/Jan/2014:05:58:35 +0900] "GET /index.htm HTTP/1.1" 200 83
```

このスクリプトでは、" "でくくられているHTTPリクエスト行から、ファイル名を取得します。HTTPのリクエストにはさまざまなメソッドがあるのですが、ここではGETメソッドで取得されたアクセスのみを集計する仕様とします。よく見られる**HTTPメソッド**について、次の表に示しました。これらはRFC 2616で定義されています。

よく見られるHTTPメソッド

メソッド名	利用シーン
GET	URIで指定されるファイルの取得
HEAD	HTTPヘッダのみを取得し、メッセージボディは取得しない
POST	メッセージの投稿などを行う
PUT	ファイルの送信を行う
CONNECT	SSL通信を行う際などに、プロキシにトンネルを要求する

❶ではまず対象のログファイルの存在をチェックします。-fは対象が通常ファイルかどうかをチェックする演算子です。それを否定演算子!と併用することで、対象がディレクトリであったり、ファイルが存在しない場合にはエラーを表示して終了するようにしています。これらファイルテストの演算子については、P.110で紹介していますのでそちらを参照してください。

❷で、ファイルごとのアクセス数をカウントしています。ここでは**GETメソッド**で取得されたリクエストのみを対象とするため、**awkコマンド**の**フィルタ**でHTTPメソッドが入る6カラム目に対して、「"GET 」とフィルタ指定しています。なおこの際、❷ではダブルクォートをエスケープしているので、実際の書き方は「$6=="¥"GET"」となっています。正常にメソッドが指定されていた場合、アクセスログ上は7カラム目にファイル名がきますから、これをawkコマンドのアクションで{print $7}として出力しています。

そして❷ではawkコマンドの出力をいったん**sortコマンド**でソートしてから、パイプの最後でもう一度、sortコマンドの**-nオプション**（数値ソート）と**-rオプション**（逆順ソート）

を実行することで、アクセスが多い順に表示されるようにしています。このsortコマンドと**uniqコマンド**の組み合わせの使い方は、P.213で紹介していますのでそちらを参照してください。このようにして、ファイルごとのアクセスログの行数、すなわちファイルごとのページビューが多い順に、ファイル名を表示することができます。

なお、ここでは「"GET 」としてフィルタをかけることで、ある程度、おかしなアクセスログを排除しようとしています。読者の皆さんが公開Webサーバを運用していると、形式が明らかにおかしい、あるいは壊れているアクセスがときおり見られることに気がつくでしょう。例えば次のようなものです。

```
xx.xx.xx.xx - - [10/Jan/2014:21:45:48 +0900] "¥x80w¥x01¥x03¥x01" 501 294
```

本来ならばHTTPの仕様上、リクエスト行の頭にはGETやPOSTなどメソッド名が付かないといけないのですが、このログはいきなり奇妙な文字列でアクセスされています。

種明かしをすると、これはhttpのポートにhttpsアクセスしようとして、SSL通信を素のhttpで流そうとするときに見られるアクセスログです。単なるクライアントの勘違いか、あるいは攻撃者が意図的に何かしらのスキャンを行っているのかもしれません。この他にも、攻撃などのために、わざとHTTP仕様上ではエラーとすべきリクエストを送りつけて来るアクセスがログに残っていることは多々あります。そのためこのサンプルでは「"GET 」とフィルタ指定することで、ある程度おかしいログは弾くように設定しました。

公開Webサーバのアクセスには実にさまざまなアクセスがやってくるため、集計レポートに多少の「ゴミ」が入ってしまうのはやむを得ないことです。このような簡単なレポート出力のスクリプトで、あまり厳密にやろうとするのは大変ですから、ある程度の妥協は必要でしょう。

注意事項

- WebサーバとしてApacheは広く使われているソフトであり、ログ解析にもさまざまなソフトが存在します。よく使われている有名なものとしては、AWStatsがあげられます。ログをグラフ化するなど視覚的にわかりやすくしてくれますので、興味のある方は利用してみてください。

関連項目

042 処理開始前に、実行権限をチェックして正常動作できることを確認してから実行する
077 Webサーバのログファイルから特定のステータスコードを返しているものだけを取得する
078 システムログからIPアドレスごとのアクセス回数を集計する

No. 080 sedでHTMLファイルの属性を置き換える際、スラッシュのエスケープが煩雑になるのを避ける

利用コマンド
sed

キーワード
スラッシュ，エスケープ

いつ使うか
sedコマンドで/（スラッシュ）を含む文字列を置換する際、いちいちエスケープしたくないとき

実行例

```
$ ./sed-slash.sh ―― 画面には何も表示されないがHTMLファイル中の/img/という文字列を/images/に置き換える
```

スクリプト

```
#!/bin/sh

# 出力ディレクトリの定義
outdir="newdir"

# 出力ディレクトリの存在チェック。なければエラー終了
if [ ! -d "$outdir" ]; then
  echo "出力ディレクトリがありません: $outdir" >&2
  exit 1
fi

# カレントディレクトリのhtmlファイルを処理
for htmlfile in *.html ―――――――――――――――――――――――― ❷
do
  # ファイル内のテキストで/img/というパスを/images/に変換する。
    sed "s%/img/%/images/%g" "$htmlfile" > "${outdir}/${htmlfile}" ―― ❸
done
```

❶

解説

このスクリプトは、カレントディレクトリにあるHTMLファイルの中の、/img/という文字列を/images/に置き換えるものです。置換後に、[newdir]というディレクトリにHTMLファイルとして出力します。利用シーンとして、Webサーバの構成変更を行い、画像を格納しているディレクトリの名前を[img]から[images]に変えた、というケースを想定しています。

sedコマンドで置換を行う際、多くの入門書ではsコマンドの後ろを/（スラッシュ）でパターンを区切るよう解説されています。この場合、置換したい文字列自身が/（スラッシュ）を含む場合は、¥記号でエスケープしなくてはいけません。

```
# 文字列 blue を red に置換
sed "s/blue/red/g" sample.txt

# 文字列 /img/ を /images/ に置換。¥/ とエスケープする
sed "s/¥/img¥//¥/images¥//g" sample.txt
```

これは非常に書くのが面倒ですし、見た目にも置換文字列がわかりにくいため、避けたい書き方です。

実は、sedコマンドで置換を行う際、**パターンを指定する区切り文字は/（スラッシュ）である必要はなく**、何でもかまいません。sの直後にある文字を自動的に区切り文字とみなしてくれるのです。ただし、あまり好き勝手な文字を指定すると別の人が読む際にわかりにくいため、慣例的によく使われる%を指定しておくのがよいでしょう。

サンプルでは、まず❶で置換後のHTMLファイルの出力ディレクトリが存在するかどうかを確認しています。**-d**は対象がディレクトリかどうかをチェックする演算子です。それを否定演算子!と併用することで、出力先が存在しなかったりディレクトリでない場合には、エラーを表示して終了するようにしています。これらファイルテストの演算子については、P.110で紹介していますのでそちらを参照してください。

❷で、forループのinに「*.html」と指定することで、カレントディレクトリのHTMLファイルを順にシェル変数htmlfileとして処理しています。❸のsedコマンドが、区切り文字を%と指定した書き方です。その出力先を、はじめにシェル変数outdirで定義したディレクトリにすることで変換後のファイルを[newdir]に保存しています。

このようにして、HTMLファイルのリンク先を、一括して変更することができます。

注意事項

- このサンプル例では、単純に/img/を/images/に置き換えるだけなので、置換対象文字列がHTMLタグの中にあるかどうかなどの細かいチェックは行っていません。そのため、HTMLの地の文に「ここでは、/img/というディレクトリを…」などの文章がある場合はその内容も置換されてしまいます。

関連項目

042 処理開始前に、実行権限をチェックして正常動作できることを確認してから実行する

No. 081 右詰めにして数値を表示し、テキストで数値の表を作る

利用コマンド
grep, printf

キーワード
書式付き出力, テキスト整形, フォーマット

いつ使うか
数値カウントなどをするコマンドで、縦位置を揃えて整形したレポートを作成したいとき

実行例

```
$ ./printf.sh
     1 (app20140101.log)
    73 (app20140105.log)
   146 (app20140111.log)
    11 (info.log)
     5 (system.log)
```

スクリプト

```
#!/bin/sh

# 検索する文字列の定義
search_text="ERROR 19:"

# カレントディレクトリの拡張子.logファイルを順に処理
for filename in *.log ―――――――――――――――――――――――――❶
do
  # マッチする行数を-cオプションで取得
  count=$(grep -c "$search_text" "$filename") ――――――――――❷
  # printfコマンドで、右詰め6桁で整形して出力する
  printf "%6s (%s)¥n" "$count" "$filename" ―――――――――――――❸
done
```

解説

このスクリプトは、カレントディレクトリのログファイル(拡張子.log)から、シェル変数search_textで定義された文字列(ここでは"ERROR 19:")を検索して、マッチする行数をファイルごとに表示するものです。例えば何かのアプリケーションのログファイル群から、ログファイルごとにエラーがでている行数を数えたいときなどに役立つでしょう。

このサンプル例では、マッチ行数を出力する際、**printfコマンド**を利用してフォーマットを揃えたレポートにしていることがポイントです。ファイルからパターンにマッチする行数をカウントして表示したい場合、**grepコマンド**だけでも、行数をカウントする**-cオ**

プションを利用して実現することはできます。

grepコマンドの-cオプションでマッチ行数をカウントする

```
$ grep -c "ERROR 19:" *.log
app20140101.log:1
app20140105.log:73
app20140111.log:146
info.log:11
system.log:5
```

しかしこの形式はあまりに素っ気なく、もう少しレポートとして見やすく整形したいという場面もあるでしょう。このスクリプトは、そのようなちょっと見栄えのよいレポートを作成するケースでの利用を想定しています。

まず❶で、for文のinで*.logと指定することで、カレントディレクトリの拡張子.logのファイルを、順にシェル変数filenameとして処理します。

❷では、マッチした行数のみを表示するgrepコマンドの-cオプションを利用して、検索文字の出現行数をカウントしています。この結果を、コマンド置換$()を利用してシェル変数countに代入します。

❸が結果を整形して表示する処理です。ここでは書式付きで文字列を出力できるprintfコマンドを利用しています。printfとは、PerlやC言語などでもよく使われる関数です。printfコマンドはこの関数をシェルから直接利用できるように、コマンド化したものです。

printfコマンドは、**書式指定子**と呼ばれる文字列を指定することでさまざまなフォーマットを設定でき、それを後ろに指定した引数(❸では$count、$filename)を代入して表示できます。ここでは$countにマッチした行数、$filenameにファイル名が入っていますから、これを整形して表示しています。

printfコマンドでよく使われる書式指定の例を次にあげておきました。

printfでよく使われる書式指定

書式指定の例	意味	出力例
"[%s]"	文字列をそのまま	[ABC]
"[%5s]"	右詰めで5桁のスペースに文字列	[ABC]
"[%-5s]"	左詰めで5桁のスペースに文字列	[ABC]
"[%-.2s]"	左詰めで頭から2文字のみ	[AB]
"[%5d]"	右詰めで5桁のスペースに整数	[123]
"[%05d]"	右詰めで5桁のスペースに整数(頭に0を付ける)	[00123]
"[%-5d]"	左詰めで5桁のスペースに整数	[123]

書式に指定する**%s**は文字列、**%d**は整数を意味します。%5sのように数字を挟むと、その桁数だけ出力スペースが確保されます。この場合には右詰めされますが、%-と指定すると左詰めされて表示されます。printfには、この他にも多くの書式指定子があります。

詳しく知りたい方は、C言語のprintfのマニュアルを「man 3 printf」として読んでみてください。

❸では、書式指定子として「"%6s (%s)¥n"」を指定しています。これより、1番目の引数(マッチ行数)を6桁右詰めで、2番目の引数(ファイル名)をカッコの中に入れて表示できます。なお、printfコマンドは改行を出力しないため、改行させるには明示的に\nが必要です。こうして実行例のように、数値位置が綺麗に揃ったレポートとして表示できるのです。

このように、整形してレポート表示したい場合はprintfコマンドを使うと便利です。読者のニーズにあわせてサンプルの書式指定子を設定してみてください。

注意事項

- manでマニュアルを表示する際、printfは関数名としての意味と、コマンド名としての意味があり、それぞれのマニュアルがあります。関数名としてのprintfのマニュアルは「man 3 printf」で、コマンド名としてのprintfのマニュアルは「man 1 printf」で、それぞれ表示できます。
- C言語でいうところのsprintf関数を使いたい場面、つまりフォーマット文字列を表示するのではなく変数に代入したい場合があります。そのようなケースでは以下のように、単純にprintfコマンドの出力を、コマンド置換を利用して変数に代入すればよいでしょう。

```
format_string=$(printf "%6s (%s)\n" "$count" "$filename")
```

No. 082 決まった桁数の数字にハイフンを入れる（郵便番号など）

利用コマンド

getopts, shift, awk, grep, sed

キーワード

オプション, 桁数, スペース除去, 郵便番号

いつ使うか

郵便番号や電話番号などのテキストから、決まった桁位置にハイフンを入れたり削除したりしたいとき

実行例

```
$ cat number-nohyphen.txt
5620001
 2250022                    ┐
A1200B1                     ┘ 数値のみの郵便番号
$ ./num-hyphen.sh number-nohyphen.txt
562-0001                    ┐
225-0022                    ┘ ハイフンが付加された
$ cat number-hyphen.txt
562-0001                    ┐
325-10022                   │
 362-0001                   ┘ ハイフン付き郵便番号
A1B-0C1C
$ ./num-hyphen.sh -d number-hyphen.txt
5620001                     ┐
3620001                     ┘ ハイフンが削除された
```

スクリプト

```
#!/bin/sh

# ハイフンを削除するかどうかのフラグ。1 ならば削除する
d_flag=0 ──────────────────────────────❶

# getopts コマンドで、削除オプション (-d) 指定を判別
while getopts "d" option ──────────────❷
do
  case $option in
    d)
      d_flag=1
      ;;
    ¥?)
      exit 1
      ;;
```

```
  esac
done

# コマンドライン引数で指定された郵便番号ファイルを、
# シェル変数filenameに代入する
shift $(expr $OPTIND - 1)                    ❸
filename="$1"

# 指定された郵便番号ファイルの存在チェック
if [ ! -f "$filename" ]; then
  echo "対象のファイルが存在しません：$filename" >&2
  exit 1                                     ❹
fi

# d_flagが指定されていればハイフンを削除、指定なければハイフン付加
if [ "$d_flag" -eq 1 ]; then                 ❺
  # *ハイフンを削除する
  # awkで前後スペース除去 → フォーマットチェック → ハイフン削除
  awk '{print $1}' "$filename" | grep '^[0-9]¥{3¥}-[0-9]¥{4¥}$' | sed "s/-
//"                                         ❻
else
  # *ハイフンを付加する
  # awkで前後スペース除去 → フォーマットチェック → ハイフン付加
  awk '{print $1}' "$filename" | grep '^[0-9]¥{7¥}$' | sed "s/¥(...¥)/¥1-/"
fi                                           ❼
```

解説

このスクリプトは、7桁の数値が書かれたファイルの3桁目と4桁目のあいだにハイフンを入れたり、ハイフン付きの7桁の数値からハイフンを削除して表示するものです。スクリプトにオプションを指定しない場合はハイフンを入れて、-dオプションを指定するとハイフンを削除します。郵便番号が書かれたファイルを操作することを想定していますが、他にも似たような数値テキスト処理に応用ができるでしょう。

このスクリプトでは、number-nohyphen.txtというハイフンが入っていない数値7桁（例：5620001）のファイルと、number-hyphen.txtというハイフンが入っている数値3桁-数値4桁（例：562-0001）のファイルを扱うと仮定します。具体的なファイルの中身は、「実行例」のcatコマンドで中身を表示している箇所を参照してください。

また、このスクリプトでハイフンを付加する際には、以下のような仕様でテキストファイルを操作することにします。

1） 各行の前後にスペースが入っている場合は、自動的にスペースを除去する
2） 7桁の数値でなければ、フォーマットエラー行として無視する

一方、ハイフン削除の際には、以下のような仕様でテキストファイルを操作することにします。

1） 各行の前後にスペースが入っている場合は、自動的にスペースを除去する
2） 「3桁数値-4桁数値」となっていなければ、フォーマットエラー行として無視する

このようなスクリプトでは、入力テキストファイルの中でフォーマットにあわない行をどうするかという問題がありますが、ここでは上記のように単純に無視することとしています。

スクリプトでは、まず❶で、オプション指定のフラグ変数d_flagを定義しています。これは-dオプション（このスクリプトではハイフンを削除する）が指定されたかどうかを判断する変数です。❷で**getoptsコマンド**（→P.3）を利用して、起動時に-dオプションが指定されたかどうかを判断して、指定されていたら1をシェル変数d_flagに代入しています。

❸では、まず位置パラメータから、コマンドライン引数で指定されたオプションを**shiftコマンド**（→P.5）で追い出します。これで位置パラメータ$1に、コマンドライン引数で指定された郵便番号ファイル名が格納されていますから、この$1をシェル変数filenameに代入しています。

❹は、指定された郵便番号ファイルが存在するかどうかの確認をしています。-fは対象が通常ファイルかどうかをチェックする演算子です。それを否定演算子!と併用することで、対象がディレクトリであったり、ファイルが存在しない場合にはエラーを表示して終了するようにしています。これらファイルテストの演算子については、P.5を参照してください。

❺で、ハイフンを削除するのか付加するのかをif文で判断して分岐しています。-dオプションが指定されていた場合はシェル変数d_flagに1がセットされていますから、この場合にはハイフンを削除する❻の処理に移ります。-dオプションが指定されていなければ、d_flagの値は0ですから、if文は偽となりハイフンを付加する❼の処理に移ります。

❻では、まず前後のスペースを除去するために**awkコマンド**で1カラム目を{print $1}として表示しています。テキストの行前後にスペースがある際、このようにすると前後のスペースを簡単に除去できます。

続いて❻でawkコマンドからパイプでつないだ後に、grepコマンドでフォーマットチェックを行っています。ここでは量指定子¥{3¥}を利用しています。量指定子とは、直前のパターンの出現回数を表現する正規表現で、次の例のように¥{¥}で囲まれた数値の回数だけ直前の文字が繰り返されるときにマッチします。

```
grep 'e¥{2¥}'              → eeにマッチ。grep "ee"と書くのと同じ
grep '[a-zA-Z]¥{8¥}'       → 英字8文字にマッチ
```

ここではまず、[0-9]として数字を意味する文字クラスを指定して、その後ろに¥{3¥}と3回出現するパターンをハイフンでつなぎ、「[0-9]¥{3¥}-[0-9]¥{4¥}」としています。これは日本語で書くと、「数値3桁の後ろにハイフンが付き、さらにその後ろに数値4桁」とい

うパターンです。郵便番号を正規表現でこのように表現しています。これにマッチしない行は、フォーマットエラーですから表示されず、つまり無視されることになります。

❻の最後に、**sedコマンド**でハイフンを削除しています。sedコマンドで文字を削除するには、キャラクタを指定して空文字列で置換すればよいので、"s/-//"とすればハイフンが削除できます。

一方、-dオプションが指定されていなければ❼の処理に分岐します。この場合はハイフンを付加する処理を行います。まず❻と同様に前後のスペースを除去するためにawkコマンドで{print $1}として、続いて「[0-9]¥{7¥}」というフォーマットチェックをgrepコマンドで行っています。つまり、数値7桁以外の、郵便番号としておかしい入力は無視することになります。

❼の最後に、sedコマンドでハイフンを付加しています。ここでは7桁の数値文字列が渡されますから、「先頭3文字の後ろにハイフンを付ける」という処理をしています。まず先頭の任意の3文字を、後方参照できるようにカッコ付きでマッチして「¥(...¥)」と指定しています。この後ろにハイフンを付ければよいわけですから、置換後文字列は後方参照¥1を利用して「¥1-」に置換すればよいわけです。

なお、このようなsedコマンドでの**後方参照**についてはP.47で説明していますので詳しくはそちらを参照してください。

このようにして、数値文字列に対して郵便番号としてのフォーマットチェックを行い、ハイフンの削除・付加を行うことができます。他にも電話番号など、読者の利用シーンにあわせてスクリプトを修正して利用してみてください。

注意事項

- このサンプルでは、入力する郵便番号ファイルはフォーマットチェックして、エラー行は無視しています。しかし場合によっては無視せずにエラーとして表示したり、あるいは重要なシステムならば「1行でもエラーがあればそこでスクリプト自体を終了させる」などの運用も必要でしょう。入力ファイルのエラー行の扱いは、事前にきちんとした設計が必要です。

関連項目

001 コマンドオプションの処理をする
018 HTMLファイルから、タグの中に書かれたコマンドを抜き出してそのまま実行する
042 処理開始前に、実行権限をチェックして正常動作できることを確認してから実行する

No. 083 ファイルサイズを減らすために、JavaScriptファイル(jsファイル)から空行を除去する

利用コマンド
sed

キーワード
空行, 軽量化, 転送量, 圧縮

いつ使うか
モバイル向けサイトなど、転送量を少しでも小さくしたいWebサイトでファイルを軽量化したいとき

実行例

```
$ ./delline.sh
$ ls newdir/
sample.js  test.js
```

ディレクトリnewdirに、カレントディレクトリのjsファイルから空行を除いたものが出力された

スクリプト

```
#!/bin/sh

# 変換後のファイル出力用ディレクトリ名
outdir="newdir" ❶

# ファイル出力用ディレクトリのチェック
if [ ! -d "$outdir" ]; then
  echo "Not a directory: $outdir"
  exit 1
fi ❷

for filename in *.js ❸
do
  # 空行およびスペースやタブのみの行を、sedコマンドのdで削除
  sed '/^[[:blank:]]*$/d' "$filename" > "${outdir}/${filename}" ❹
done
```

解説

このスクリプトは、カレントディレクトリにあるJavaScriptのファイル(拡張子.js)から、空行およびタブ・スペースのみの行を削除して、ファイルサイズを軽量化するものです。モバイル端末向けのサイトなどで、少しでも転送量を小さくしたいケースを想定しています。

❶では、まず変換後の出力用ディレクトリを定義しています。ここでは複数のjsファイ

ルを処理しますので、それらをシェル変数outdirで定義したディレクトリに出力します。

❷で、この出力用ディレクトリのチェックをしています。**-d**は対象がディレクトリかどうかをチェックする演算子です。それを否定演算子!と併用することで、出力先が存在しなかったりディレクトリでない場合には、エラーを表示して終了するようにしています。これらファイルテストの演算子については、P.110で紹介していますのでそちらを参照してください。

❸で、for文のinに「*.js」と指定することで、カレントディレクトリのJavaScriptファイルを順にシェル変数filenameとして処理しています。❹の**sedコマンド**が、空行およびタブ・スペースのみの行を削除して出力する例です。ここではsedコマンドに「/<パターン>/d」と、"d"を指定することで、パターンにマッチする行の削除を行っています。

削除するパターンに使っている[[:blank:]]とは、**POSIX文字クラス**を利用した書き方です。ここで、[:blank:]というのはスペース記号やタブ記号など、いわゆる「空白」に相当するものを意味する文字集合です。これをさらに[]でくくることで、空白記号の文字クラスを指定しています。後ろに*が付いていますから、これは全体では「行頭の次に、0回以上の空白文字が続き、行末となる」という行にマッチします。これはつまり空行およびタブ・スペースのみの行ですから、このsedコマンドの出力結果をリダイレクトすることで、不要行のみ削除した結果が得られます。ここでは出力先を、はじめにシェル変数outdirで定義したディレクトリにすることで、変換後のファイルをディレクトリnewdirに保存しています。

なお、このサンプルのようにJavaScriptのファイルサイズを小さくするには、専用のツールも多くあります。ファイルサイズに厳しいモバイルサイト運用の際には、一般的にはそのような軽量化ツールが利用されますが、ここではシェルスクリプトで出力するサンプル例を紹介しました。

注意事項

- このスクリプトは、ファイルの改行コードがLF (¥n) であることを前提としています。CRLF (¥r¥n) の場合には、空行にマッチしません。
- 軽量化ツールでは、空行を消すだけでなく、もっとアグレッシブに改行も消してしまうものがあります。そのようなコードにしたければ、❹の部分を次のように変えて、trコマンドで改行を消すことができます。

```
sed '/^[[:blank:]]*$/d' "$filename" | tr -d '¥n' >
"${outdir}/${filename}"
```

関連項目

042 処理開始前に、実行権限をチェックして正常動作できることを確認してから実行する

No. 084 テキストファイルからHTMLファイルを作る

利用コマンド
sed

キーワード
HTML, エスケープ, 文字参照, 改行, テキストファイル

いつ使うか
サーバ上のテキストファイルを簡易的にHTMLファイルに変換し、Webブラウザで閲覧したいとき

実行例

```
$ cat sample.txt
1 + 1 < 3
"Hello!"
A & B
$ ./txt2html.sh sample.txt > sample.htm
$ cat sample.htm ―――――――――――――― 簡易HTMLを確認
1 + 1 &lt; 3<br>
&quot;Hello!&quot;<br>
A &amp; B<br>
```

スクリプト

```
#!/bin/sh

# HTMLとしてエスケープが必要な記号を文字参照に置き換え、
# 最後に行末を<br>タグに置換する。
sed -e 's/&/¥&amp;/g' ¥ ――❶
-e 's/</¥&lt;/g' ¥
-e 's/>/¥&gt;/g' ¥
-e "s/'/¥&#39;/g" ¥
-e 's/"/¥&quot;/g' ¥
-e 's/$/<br>/' ¥ ――――――――❷
"$1"
```

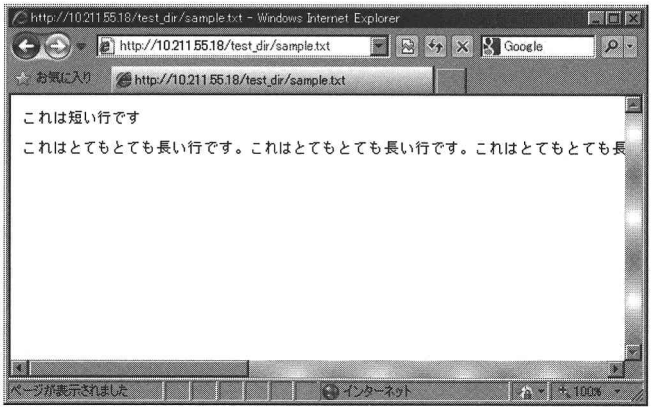
解説

このスクリプトは、テキストファイルを簡易的なHTMLファイルに変換して出力するものです。サーバ上にあるテキストファイルを、Webブラウザで手軽に見たいケースを想定しています。

リモートホストで作業中に、接続先にあるマニュアルなどのテキストファイルを、狭い端末内ではなくWebブラウザで閲覧したい場合があります。この際、長い行があるとInternet Explorerなどでは自動的な折り返しがされず、横スクロールバーが表示されて

非常に見にくいことがあります。

ブラウザでテキストファイルを表示すると読みにくくなる



HTMLファイルならば長い行も自動的に折り返しされますから、このような場合は、テキストファイルをブラウザで表示できるようにHTMLファイルにできれば便利です。そのためこのサンプル例で、テキストファイルを簡易HTMLファイルに変換してみましょう。

テキストファイルをHTMLファイルにするには、厳密にやるならば<html>タグや<head>タグを組み立てたりと、さまざまな準備が必要です。しかしここでは、手元のブラウザで見られれば十分ですから、単に行末に
を付けるだけというシンプルな仕様にしてみます。ただしこの際、不用意にJavaScriptなどのコードが実行されないよう、HTMLとしてエスケープが必要な記号は正しくエスケープ処理します。これはセキュリティ対策のための処理でもあります。

利用者がWebブラウザで閲覧するプログラムに外部からの入力値がある場合には、その外部入力は、悪意のある攻撃者によって「汚染されている」可能性があることを常に意識しなくてはいけません。例えばJavaScriptは単なるテキストファイルとして見るぶんには何も問題がありませんが、Webブラウザに解釈させた途端に、危険な攻撃コードとして実行されてしまう可能性があります。ですから、誰かが書いたテキストファイルをWebブラウザで表示するこのようなプログラムでは、**HTMLタグをエスケープして無害化する処理**が重要です。なおPHPやRubyなど一般的なスクリプト言語では、このために専用のエスケープ関数が用意されています。これは「注意事項」にあげてあります。

❶では、**sedコマンド**で複数のパターンをまとめて置換しています。このように複数のパターンを指定する際は、❶のようにsedコマンドの**-eオプション**でパターンを続けて指定してつなげれば、パイプ処理を使わずに書くことができます。

なお❶では、HTMLとしてエスケープが必要な記号を文字参照に置き換えています。この際に置き換えている記号は次のとおりです。

ブラウザで表示させるための置換

元の記号	置き換え文字
&	&
<	<
>	>
'	'
"	"

シングルクォートとダブルクォートを置き換える際は、sedコマンドの全体をくくるクォート記号に注意が必要です。サンプル例では基本的にすべてをシングルクォート記号でくくっていますが、シングルクォート記号の置換「"s/'/¥'/g"」だけは置換対象の'(シングルクォート記号)をわざわざエスケープしなくてもよいように、全体をダブルクォート記号でくくっています。このようなクォート記号の混在の際の注意については、P.39で説明していますのでそちらを参照してください。

❷の置換パターンは、行末に
タグを付加する処理です。シェルスクリプトでテキストの行末部分に特定の文字列を追加するにはさまざまな手法がありますが、サンプル例ではこれにもsedコマンドを利用しています。sedコマンドでは$が行末を表しますので、これを
で置換することで行末に
タグが付加できます。これが「's/$/
/'」という置換パターンです。

このスクリプトは単にsedコマンドの置換パターンをつなげて並べているだけですが、このような処理をいちいち手打ちでやっていては大変ですし、処理の漏れが起こるかもしれません。このようにスクリプト化して、自動化できるようにしておくとよいでしょう。

注意事項

- このスクリプトで作られるHTMLファイルは<html>タグや<head>タグがないなど、あまりHTML的に正しい書き方にはなっていません。テキストを簡易的にWebブラウザで見るための事例、とご理解ください。
- 参考のため、他のスクリプト言語でHTMLタグをエスケープする関数を次にあげておきます。

HTMLをエスケープする関数

言語	関数名
PHP	htmlspecialchars
Perl	escapeHTML(CGI.pmモジュールを利用)
Ruby	CGI.escapeHTML

関連項目

015 シングルクォートの中でシングルクォートを使う

No. 085 HTMLファイルの文字コードを自動的に判別して、UTF-8でエンコードされたファイルに変換する

利用コマンド
grep, sed, iconv

キーワード
HTML, metaタグ, 文字コード

いつ使うか
HTMLファイルのmetaタグから自動的に文字コードを判別し、UTF-8へ変換したいとき

実行例

```
$ ./charset-utf8.sh
$ ls newdir/                    ディレクトリnewdirにhtmlファイルが出力されたことを確認
index.html   sjis.html
```

スクリプト

```
#!/bin/sh

# 変換後のファイル出力先ディレクトリ名
outdir="newdir"                                          ❶

# ファイル出力先ディレクトリのチェック
if [ ! -d "$outdir" ]; then
  echo "Not a directory: $outdir"                        ❷
  exit 1
fi

# カレントディレクトリの .html ファイルを対象
for filename in *.html                                   ❸
do
  # grepコマンドでmetaタグのContent-Type行を選択し、
  # sedコマンドでcharset=指定部分を抜き出す
  charset=$(grep -i '<meta ' "$filename" |¥
  grep -i 'http-equiv="Content-Type"' |¥                 ❹
  sed -n 's/.*charset=¥([-_a-zA-Z0-9]*¥)".*/¥1/p')

  # charsetが取得できていない場合は、iconvコマンドを実行せずにスキップする
  if [ -z "$charset" ]; then
    echo "charset not found: $filename" >&2              ❺
    continue
  fi
```

```
  # metaタグから取り出した文字コードから、UTF-8へと変換し、
  # ディレクトリ$outdirに出力する
  iconv -c -f "$charset" -t UTF-8 "$filename"  > "${outdir}/${filename}"  ❻
done
```

解説

このスクリプトは、カレントディレクトリにあるHTMLファイル(拡張子.html)を、HTMLファイル内のmetaタグで指定されている文字エンコードを利用して、UTF-8に変換するものです。さまざまな文字コードのHTMLファイルを、一括してUTF-8に変換するケースを想定しています。変換には、文字コードを変換する標準的なコマンドである**iconvコマンド**を用います。

❶で、まず変換後の出力先ディレクトリを定義しています。ここでは複数のhtmlファイルを処理しますので、それらをシェル変数outdirで定義したディレクトリに出力します。

❷で、この出力先ディレクトリのチェックをしています。**-d**は対象がディレクトリかどうかをチェックする演算子です。それを否定演算子**!**と併用することで、出力先が存在しなかったりディレクトリでない場合には、エラーを表示して終了するようにしています。これらファイルテストの演算子については、P.110で紹介していますのでそちらを参照してください。

❸で、for文のinに「*.html」と指定することで、カレントディレクトリのHTMLファイルを順にシェル変数filenameとして処理しています。for文の中では、❹でHTMLファイルから文字コードを取り出し、シェル変数charsetに格納しています。なお❹は、各コマンドをパイプでつないでおり1行が長いため、行末に¥を置いて見かけの改行をしています。

ここで、HTMLのcharset指定についておさらいしておきましょう。HTMLではファイルの**文字コード**を、metaタグの中のcontent属性内で、**charset=**を指定して行います。

HTMLファイルでの文字コード指定例

・EUC指定

```
<meta http-equiv="Content-Type" content="text/html; charset=euc-jp">
```

・シフトJIS指定

```
<meta http-equiv="Content-Type" content="text/html; charset=Shift_JIS">
```

・content属性とhttp-equiv属性の順序は逆になることもある

```
<meta content="text/html; charset=utf-8" http-equiv="Content-Type" />
```

・タグが大文字で書かれたり、charsetの前のスペースはないこともある

```
<META HTTP-EQUIV="Content-Type" CONTENT="text/html;charset=UTF-8">
```

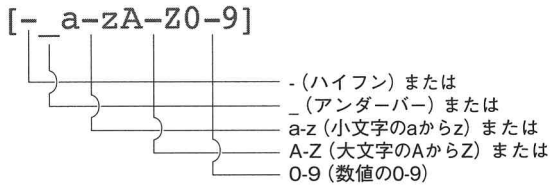
上記のように、HTMLでは表記の揺れもありすべてに対応するのは大変です。このスクリプトでは、❹で以下のような仕様で文字コード指定を抽出してみました。

(1)「<meta」と「http-equiv="Content-Type"」を含む行をgrepで選択
(2) charset= という文字列の、後ろのイコールで指定されている文字列をsedコマンドで抽出

❹では、コマンド群全体を**コマンド置換$()**でくくって、シェル変数charsetに代入しています。はじめの2つの**grepコマンド**では、**-iオプション**を付けて大文字小文字を無視しています。これはタグが大文字で書かれることもあるためです。

❹の最後のパイプ処理で**sedコマンド**が行っているのは、「charset=xxxxx」の「xxxxx」という文字列を取り出す処理です。ここではcharsetのイコール記号の後ろに取り出すべき文字コードが入っていますから、それを「charset=¥([-_a-zA-Z0-9]*¥)」とパターン指定しています。文字コードはEUC-JPなど基本的に「アルファベット、ハイフン、アンダーバー、数値」のみからなりますから、これを文字クラスとして[-_a-zA-Z0-9]として表現しています。

文字コードを特定するための文字クラス



このときハイフンは範囲指定にも使われるため、文字クラスでハイフンそのものを指定する際には先頭か最後に置かなければなりません。

sedコマンドのマッチ部分で、charset=の後ろを¥(¥)でくくっているのは、**後方参照**(マッチ部分取り出し)できるようにするためです。sedコマンドを、パターンスペースを出力しない**-nオプション**を用いて、同時に置換後の文字列を後方参照「**¥1**」としてpフラグで出力することにより、マッチした部分のみ取り出せます。なお、このようなsedコマンドでの後方参照と-nオプションでのマッチ部分取り出しについては、P.46を参照してください。

❺では、❹で取り出したcharsetの文字列をチェックしています。charset指定がないHTMLファイルなどで、文字コードの判別に失敗した場合、シェル変数charsetには空文字列が入っています。そのため❺では**testコマンド**の空文字列かどうかを調べる**-z演算子**を用いてシェル変数charsetをチェックし、これが真だった場合はcontinue文で次のループへとスキップしています。

さて、❹によりシェル変数charsetに文字コードが代入できたので、これをiconvコマンドで処理します(❻)。iconvコマンドの使い方は次のようになります。

書 式 iconvコマンドの書式

```
iconv -f <入力文字コード> -t <出力文字コード> <ファイル名>
```

入力文字コードは❹で取り出せましたから、これを-fオプションに指定しています。出力文字コードは、このサンプルではUTF-8決め打ちとしています。また❻では、指定された文字コードで想定しないコードがあった場合には、その文字を無視するようにする-cオプションを加えています。この-c指定をしないと、1文字でもゴミが入り込んでいるHTMLファイルがエラーで止まってしまいますので、実用上、入れておいたほうがよいでしょう。

❻の最後に、出力をディレクトリnewdirにリダイレクトして、変換後のファイルを出力しています。こうして、一括してHTMLファイルをUTF-8で出力できます。

なおHTMLファイルは歴史が長いことから表記にもいろいろと揺れがあり、このスクリプトでうまく扱えないファイルがあるかもしれません。読者の環境にあわせて、grepやsedコマンドのパターン部分は適宜修正してください。

注意事項

- このスクリプトでは、charset指定がされていないHTMLファイルは扱えません。また出力されるファイルは、metaタグで何が指定されていてもUTF-8のファイルとして出力されます。
- HTMLの新しい規格「HTML5」では、次のような省略したcharset指定ができます。

```
<meta charset="UTF-8">
```

サンプル例ではこの形式には対応していません。この場合、sedコマンドの部分を、次のように変えてみてください。

```
sed -n 's/<meta .*charset="¥([-_a-zA-Z0-9]*¥)".*/¥1/p'
```

- HTMLの古い形式では、Shift_JISは頭にx-を付けて「x-sjis」と表現していました。iconvではこの形式に対応していないためエラーとなります。

関連項目

018 HTMLファイルから、タグの中に書かれたコマンドを抜き出してそのまま実行する

CHAPTER

07

シェルの機能を使いこなす

この章では、シェルの機能であるシグナル処理や並列処理、別のコマンドを実行するラッパーの書き方、外部コマンドの扱い方などを紹介します。この章のサンプル例を活用すれば、ただ順番にコマンドを並べるだけの書き方から、割り込みや並列実行などの非同期処理を利用した、一歩進んだシェルスクリプトが書けるようになるでしょう。

No. 086 関数やif文内などでヒアドキュメントを使う際、ベタ書きせずに行頭にタブを入れて見やすくする

利用コマンド
cat

キーワード
ヒアドキュメント, インデント, タブ

いつ使うか
関数ブロックやif文など、インデントが必要なところで、ヒアドキュメントもインデントしてコーディングしたいとき

実行例

```
$ ./here-indent.sh "My Title"
<html>
<head>
  <title>My Title</title>
</head>

<body>
  <p>Auto HTML sample.</p>
</body>
</html>
```

ヒアドキュメント本体は行頭にタブが付いているが、出力にはタブが付いていない

スクリプト

```
#!/bin/sh

# コマンドライン引数のチェック
if [ -z "$1" ]; then
    echo "title要素を引数で指定してください" >&2
    exit 1
else
    # コマンドライン引数 $1 の文字列をtitle要素に入れて表示。
    # ヒアドキュメントに- (ハイフン) 指定して、
    # 行頭タブを無視してインデントしている
    cat <<-EOT
    <html>
    <head>
     <title>$1</title>
    </head>

    <body>
     <p>Auto HTML sample.</p>
```

❶
❷

```
    </body>
    EOT
    # ※↑このヒアドキュメント部分のインデントは、スペースではなくタブである
fi
```

解説

このスクリプトは、コマンドライン引数で与えられた文字列を<title>要素として、HTMLファイルを出力するものです。if文の中で、**ヒアドキュメントをインデント**していることがポイントです。なお、ヒアドキュメントについての基本的な書き方は、P.56で説明していますのでそちらを参照してください。

ヒアドキュメントを使う際に不便なのが、インデントができないということです。つまり、ヒアドキュメント内のテキストは行頭のタブを含めてすべてそのまま文字列として認識されてしまいますから、読みやすくするためにインデントを入れてコーディングすると実行時に余計なタブが入ってしまいます。これでは、関数内やif文の中でインデントをせずにベタ書きをしなくてはいけないため、見た目にもわかりやすくありません。

リスト1 インデントせずに書くとスクリプトはわかりにくくなる

```
if [ -z "$1" ]; then
    echo "title要素を引数で指定してください" >&2
    exit 1
else
   cat << EOT
<html>
<head>
(省略)
</body>
EOT
# ↑このヒアドキュメントはインデントできない
fi
```

しかしヒアドキュメントには、**<<-**のように-（ハイフン）を利用することで、行頭タブを無視させることのできる記法があります。本サンプルでは、この記法によりヒアドキュメントのインデントを行っています。

まず❶で、**testコマンド**の「空文字列ならば真」となる**-z演算子**を利用して引数チェックを行っています。これが真ならば引数が指定されていませんから、エラーを表示して終了しています。

❷で、<title>要素を指定したHTMLファイルをcatコマンドで出力しています。本来ならばこのようなif文の中などのヒアドキュメントは、余計な行頭タブが入らないようにベタ書きしないといけません。しかし❷では、ヒアドキュメントの指定（<<）の後ろに、-（ハ

イフン)を付けています。この指定を行うと、シェルはヒアドキュメント部分の行頭のタブを無視します。結果として、サンプル例のようにヒアドキュメント本体をタブでインデントすることができるようになります。

このようにヒアドキュメントをインデントして書くと、わかりやすく読みやすいコーディングを行うことができますから、保守性の向上にもつながります。関数内やif文ブロック内などでヒアドキュメントを用いる場合は、この「<<-」の記法を利用してみてください。

注意事項

・この記法で無視できるのは行頭タブのみで、行頭スペースは無視されません。そのためこの記法を利用してコーディングする際、インデントはスペースではなくタブで行ってください。vimやEmacsなど多くのエディタでは、インデントをスペースで行うかタブで行うかの設定と、タブ幅の設定ができるようになっています。

関連項目

022 ヒアドキュメントで変数展開をせずにそのまま$strのように表示する

No. 087 スクリプト実行中にシグナルを受け取って、現在の実行状態を出力する

利用コマンド

trap, nc, sleep

キーワード

シグナル, 割り込み, トラップ

いつ使うか

長い処理時間のかかるシェルスクリプトを実行中に、プロセスを停止せずに内部の実行状態を表示したいとき

実行例

```
$ ./sig-usr1.sh
Connection to 192.168.2.105 80 port [tcp/http] succeeded!
Connection to 192.168.2.105 80 port [tcp/http] succeeded!
Connection to 192.168.2.105 80 port [tcp/http] succeeded!
Try Count: 3
Connection to 192.168.2.105 80 port [tcp/http] succeeded!
  :
```

他の端末からシグナルを送ることにより現在のTry Countを表示できた

スクリプト

```
#!/bin/sh

# 実行回数カウンタ
count=0 ❶
# 通信テストの対象サーバ
server="192.168.2.105" ❷

# シグナルUSR1にトラップを設定。現在の$countを表示する
trap 'echo "Try Count: $count"' USR1 ❸

# ncコマンドでの連続通信テストのループ
while [ "$count" -le 1000 ] ❹
do
  # カウンタを1増やし、ncコマンドで通信テストを行う
  # 最後に1秒のウェイトを入れる。
  count=$(expr $count  + 1)
  nc -zv "$server" 80 ❺
  sleep 1
done
```

解説

このスクリプトは、繰り返し処理により実行時間が長いスクリプトを実行中に、該当プロセスに**killコマンド**で**USR1シグナル**を送ることで、現在までの実行回数を表示させるものです。ここでは、**ncコマンド**でTCPポート80 (http) 宛てに長時間の繰り返し通信テストを行い、シグナルが送られるとその都度、現在までのテスト回数を"Try Count:"として表示するケースを想定しています。

このサンプルで利用しているncコマンドは、サーバ宛ての通信をテストできる便利なコマンドです。使い方については、P.163を参考にしてください。

実行中のプロセスの状態を知るには、プロセスにシグナルを送ってそのタイミングで内部の状態を表示するという手法があります。シェルスクリプトでこのようなシグナル処理をするためには、trapコマンドを用います。なお、**trapコマンド**とシグナルの関係は、P.6で詳しく説明していますのでそちらを参照してください。

サンプル例で用いているUSR1シグナルは、アプリケーションごとに独自に動作を定めることのできるシグナルです。例えばWebサーバとしてよく使われるApache httpdでは、USR1シグナルを送るとgracefulモードで「緩やかに再起動」しますし、Linuxのddコマンドに USR1シグナルを送ると、コマンドを続行させたまま「現在までに処理したブロック数」を途中表示させることができます。

このような仕組みをシェルスクリプトにも実装しておくと、長い処理時間がかかるスクリプトで、実行中の状態を「覗く」ことができて便利です。サンプルではncコマンドでテストを行った回数を表示することにしてみました。

まず❶と❷で、テスト回数のカウンタ初期化と、テスト対象サーバを定義しています。シェル変数countが、実行テスト回数になります。

❸が、シグナルを受信した際の処理を記述している部分です。trapコマンドでUSR1シグナルを受け取り、その処理として"Try Count: $count"をechoコマンドで表示することで現在までのテスト回数が表示できます。この❸の部分は、スクリプトとして上から順に実行されている時点では何も表示されません。実行中に実際にUSR1シグナルが送られたときに、この処理が行われるように定義をする文だと理解してください。このようにして、シェルスクリプトではtrapコマンドを利用して割り込み処理を書くことができます。

❹で、while文を用いて連続通信テスト (実行時間の長い処理) を行います。ここではカウンタを1増やしながら1000回実行することにしています。❺で、ncコマンドにより対象サーバのポート80 (http) へ通信テストを行い、sleepコマンドで1秒待ってから、exprコマンドでシェル変数countの値を1増やしていきます。

実際にこのプロセスにUSR1シグナルを送るには、スクリプトを実行させたまま別の端末ウィンドウを開いて、実行中のプロセス一覧を表示する**psコマンド**を実行してください。

psコマンドで実行中プロセス一覧を表示

```
$ ps x
  PID TTY      STAT   TIME COMMAND
29616 ?        S      0:00 sshd: user1@pts/0
29617 pts/0    Ss     0:00 -bash
29636 ?        S      0:00 sshd: user1@pts/1
29637 pts/1    Ss     0:00 -bash
29658 pts/1    S+     0:00 /bin/sh ./sig-usr1.sh
29698 pts/1    S+     0:00 sleep 1
29699 pts/0    R+     0:00 ps x
```

上記のPID欄を見ると、sig-usr1.shを実行しているプロセスIDは29658です。このプロセスIDに、killコマンドでUSR1シグナルを次のように送ります。

sig-usr1.shの進行具合を表示させる

```
$ kill -s USR1 29658
```

こうするとスクリプトはUSR1シグナルを受信し、実行例のように"Try Count"としてncコマンドの実行回数が表示されます。

注意事項

・OSによって利用できるシグナルは多少変わります。現在の環境で使えるシグナルは、killコマンドに-lオプションを付けることで表示できます。このとき表示されるシグナルはすべて先頭にSIGが付いた形式、例えばHUPシグナルは「SIGHUP」などと表示されます。

関連項目

002 キーボードから Ctrl + C 入力されたときに、現在の状態を出力してから終了する
061 あるサーバの特定ポートへ通信できるかのチェックスクリプト

No. 088 HUPシグナルを受け取って、実行中に設定ファイルを読み込みなおす

利用コマンド

trap, uptime, .（ドットコマンド）

キーワード

シグナル, 割り込み, トラップ, リロード

いつ使うか

設定ファイルを読み込むスクリプトで、実行中に設定ファイルを書き換えた後、プロセスを止めずに設定ファイルを再読み込みさせたいとき

実行例

```
$ ./sig-hup.sh
$ kill -s HUP <プロセスID>
```

スクリプトを実行したまま、setting.confを書き換え、他の端末からシグナルを送ることにより、ファイルの出力先を変えることができる

スクリプト

```
#!/bin/sh

# 環境初期化のシェル関数。ログ出力先を設定した setting.conf を読み込む
loadconf() {
  . ./setting.conf ―――❶
}

# HUP シグナルの割り込みで設定を読み込みなおすように定義
trap 'loadconf' HUP ―――❷

# 通常起動時の、はじめの初期化を行う
loadconf ―――❸

# 無限ループで実行
while : ―――❹
do
  # uptime コマンドの結果を、setting.conf で指定されたパスにログ出力
  uptime >> "${UPTIME_FILENAME}" ―――❺
  sleep 1
done
```

解説

このスクリプトは、サーバのロードアベレージ状態を1秒に1回ファイルに出力し続けるものです。**uptimeコマンド**は次のように、サーバの起動時間と、サーバの過去1分/5分

/15分のロードアベレージを出力します。この結果を長時間とることで、サーバの負荷状態を見ることができるでしょう。

uptimeコマンドで負荷状態を表示する

```
$ uptime
 21:24:45 up 53 days, 23:33,  1 user,  load average: 0.43, 0.38, 0.32
```

このサンプル例では、シェルスクリプトのプロセスに**HUP（ハングアップ）シグナル**を送ることで、プロセス終了させずに出力先のファイルを切り替えできることがポイントです。このような動作は、サーバに常駐するデーモンプログラムなどでよく使われる手法です。

このサンプルは起動時にまず❶で、外部設定ファイルsetting.confを読み込んでいます。このsetting.confとは次のように、出力ファイル名を指定する定義ファイルです。

リスト1 このサンプルで使用している設定ファイル（setting.conf）

```
UPTIME_FILENAME="/var/tmp/uptime.log"
```

このように、外部ファイルを**.（ドット）コマンド**で取り込む例は、P.41で紹介していますので詳しくはそちらを参照してください。このサンプルでは、割り込み処理により、プロセス動作中にこの設定ファイルを再度読み込ませようというわけです。シェルスクリプトで割り込みに対応する（シグナル処理をする）ためには、**trapコマンド**を用います。trapコマンドとシグナルの関係は、P.6で詳しく説明しています。

このスクリプトでは、❶でまず環境初期化のシェル変数を定義しています。これはカレントディレクトリのファイルsetting.confを読み込むだけの簡単なものです。これでシェル変数UPTIME_FILENAMEが設定され、uptimeコマンドのログ出力先が設定されます。

❷が、trapコマンドでHUPシグナルによる割り込みを定義している部分です。ここでは、設定ファイルを再読み込するためにloadconf関数を呼んでいます。trapコマンドは実際の処理をシングルクォートでくくるために長い処理が書きにくいので、このようにシェル関数を別途定義しておいて、その関数を呼ぶように記述するという手法はよく使われるやり方です。

設定ファイルを再読み込みさせるような動作には、通常はHUPシグナルが使われます。例えばWebサーバとしてよく使われるApache httpdでは、親プロセスにHUPシグナルを送ると、設定ファイルhttpd.confの再読み込みを行うreload動作をします。このような慣習から、❷でも設定ファイルの再読み込みはHUPシグナルで動作するようにしています。

❸は、通常起動した際に最初に実行される行です。ここではloadconf関数を呼んで、設定を初期化します。この時点で、シェル変数UPTIME_FILENAMEには"/var/tmp/uptime.log"が設定されることになります。

❹から、実際にuptimeコマンドを実行するwhileループになります。ここではwhileに:（ヌルコマンド）を指定して、無限ループで実行しています。❺では、uptimeコマンドの

結果をシェル変数UPTIME_FILENAMEに出力し、**sleepコマンド**で1秒待ってから、再びuptimeコマンドを繰り返すという動作になっています。

では、設定ファイルを書き換えて、プロセスが起動したままファイルの出力先を動的に変更してみましょう。まずsetting.confをテキストエディタで開いて次のように書き換えて、コマンドの結果がuptime2.logという別のファイルに出力されるように設定します。

リスト2 設定ファイルを変更する

```
UPTIME_FILENAME="/var/tmp/uptime2.log"
```

そして実際にこのプロセスにHUPシグナルを送るために、別の端末ウィンドウを開いて、psコマンドで現在実行中のPIDを調べます。例えば、sig-hup.shを実行しているPIDが29658であれば、次のように**killコマンド**の**-sオプション**でHUPシグナルを送ればよいということです。

HUPシグナルを送ると設定ファイルを再読み込みする

```
$ kill -s HUP 29658
```

こうすると、いままで/var/tmp/uptime.logに出力されていたログが、/var/tmp/uptime2.logに出るように切り替わります。このようにして、プロセスを止めずに動的に設定を変えることができるわけです。

注意事項

・このスクリプトは無限ループのため、終了させるにはkill -9コマンドを利用するか、起動した端末から Ctrl + C を入力してください。

関連項目

002 キーボードから Ctrl + C 入力されたときに、現在の状態を出力してから終了する
016 変数や関数を外部ファイルに記述する

No. 089 異常終了してもゴミが残らないよう、終了前に作業ファイルを消去して後始末を行う

利用コマンド
trap, rm

キーワード
一時ファイル, 終了, シグナル, トラップ

いつ使うか
一時ファイルを利用するスクリプトで、正常終了・強制終了ともに共通の終了処理を記述したいとき

実行例

```
$ ./sig-exit.sh
^C                                    Ctrl + C で強制終了させたが
$ ls
sig-exit.sh  calcA.sh  calcB.sh       一時ファイルは残っていない
```

スクリプト

```
#!/bin/bash

# 一時ファイルを定義し、ファイルの中を空に初期化する
tmpfile="calctmp.$$"
: > "$tmpfile"                                    ❶

# トラップの設定。終了時に一時ファイルを削除する
trap 'rm -f "$tmpfile"' EXIT                      ❷

# 長い計算を行う外部スクリプトを実行する
./calcA.sh >> "$tmpfile"
./calcB.sh >> "$tmpfile"                          ❸

# 計算結果を足し合わせて、最終数値を計算する
awk '{sum += $1} END{print sum}' "$tmpfile"       ❹
```

解説

このスクリプトは、正常に終了した場合でも、キーボードから Ctrl + C を叩かれるなどで強制終了した場合でも、一時ファイル(tmpfile)を削除して終了するものです。なお動作環境として、bashを前提にしています。

このスクリプトでは、何らかの計算を行ってその数値結果を出力する外部スクリプトcalcA.shとcalcB.shを利用しています。このスクリプトは次のように、単に計算結果を表

示するものと仮定します。

calcA.shとcalcB.shの実行結果

```
$ ./calcA.sh
3928
$ ./calcB.sh
79104
```

このサンプルでは最終的に、calcA.shとcalcB.shで出力される2つの値を足し合わせた数値を計算したい、という状況を仮定します。そして処理が長くかかるため、強制終了などされた場合にはゴミが残らないように一時ファイルを削除する仕様とします。このために、スクリプト終了時に一時ファイルを削除するコマンドを実行したいわけです。

スクリプト終了時の共通処理を書きたい場合には、trapコマンドを使ってEXITシグナルを用いるのが基本的なやり方です。**EXITシグナル**とはbashで利用できる**疑似シグナル**で、正常終了時(スクリプトが最終行まで実行された場合、もしくは**exitコマンド**で明示的に終了された場合)、強制終了時(キーボードから Ctrl + C を押されたり、killコマンドで終了させた場合)、いずれの場合にも発生します。そのため、そのEXITシグナルでの割り込み処理として、一時ファイルの削除を書いておくと、スクリプト終了時に一時ファイルのゴミを確実に消すことができます。

サンプル例では、まず❶で、計算結果を一時的に保存しておく一時ファイルのファイル名を定義しています。この際、**:(ヌルコマンド)**を用いてファイルを空にしています。ここで"calctmp.$$"というのは一時ファイルを作る際にファイル名にプロセスIDを入れるために用いられる記法です。詳しくはP.128を参照してください。また、:(ヌルコマンド)を用いたファイルの初期化についてはP.71を参照してください。

❷でEXITシグナルの設定を、trapコマンドで行っています。ここでは終了時に一時ファイルを削除したいため、**rmコマンド**で一時ファイル$tmpfileを削除するよう設定しています。この設定により、スクリプト終了時にはこのtrap設定されたコマンド行が実行されるため、一時ファイルが自動的に削除されます。

❸で、一時ファイルに計算結果を出力しています。このスクリプトでは、この計算にとても時間がかかるという仮定をしているため、この計算途中で強制終了されるケースを想定しています。

❹で、一時ファイルから最終計算を行っています。**awkコマンド**で一時ファイルの1カラム目を取り出し、sumという変数に足し合わせて表示することで最終計算をしています。

スクリプトが❹まで実行されれば、スクリプトの正常終了時にEXITシグナルが発生して一時ファイルの削除が行われます。また、計算途中にキーボードから Ctrl + C が押されるなどして強制終了された場合も、やはりEXITシグナルが発生して一時ファイルの削除が行われます。

このようにして、正常終了・強制終了に関係なく、共通した終了処理を書くことができます。一時ファイルを確実に削除したいときなどに利用するとよいでしょう。

bashの疑似シグナル

本項で利用したEXITシグナルとは、bashで利用できる疑似シグナルです。通常、シグナルはOSが発生させ管理するものですが、疑似シグナルはbashが発生させ、受け取るのもbashプロセスのみです。もっとも、シェルスクリプトを書く上ではどちらのシグナルもtrapコマンドで同等に扱えますので、それが疑似シグナルであるか「本物の」シグナルであるかは、あまり気にする必要はありません。

bashで利用できるシグナルを、次にあげておきます。例えばERRシグナルを使うと、エラーが発生するたびに特定の関数を実行するという処理を簡単に記述することができます。

bashの疑似シグナル

疑似シグナル名	発生タイミング
EXIT	スクリプトの終了時
ERR	コマンドの終了ステータスが非ゼロだった
DEBUG	コマンドが実行された
RETURN	.(ドットコマンド)またはsourceコマンドで外部スクリプトを読み込んだ

関連項目

027 ファイルの中身を消去して、ゼロバイトの空ファイルにする
049 二重起動が可能な一時ファイル作成する

No. 090 常に指定した環境変数を設定してコマンドを実行するために、ラッパースクリプトを作成する

利用コマンド
exec

キーワード
ラッパー，環境変数

いつ使うか
ラッパースクリプトを作成したいとき

実行例

```
$ ./exec.sh -o output.txt ―――― myappdが起動される
```

スクリプト

```
#!/bin/sh

# TMPDIR環境変数の設定
TMPDIR="/disk1/tmp"
export TMPDIR

# execコマンドでmyappdを実行。コマンドライン引数を"$@"で渡す
exec ./myappd "$@" ――――――――――――――――❶
```

解説

このスクリプトは、環境変数TMPDIRを"/disk1/tmp"に設定したうえで、カレントディレクトリにあるmyappdというプログラムを実行するものです。ここでmyappdとは、環境変数TMPDIRをテンポラリディレクトリとして用いるプログラムであると仮定しています。

サンプル例のように、環境変数などを独自指定したうえで他のプログラムを実行するプログラムを、**ラッパー**と呼びます。このようなラッパースクリプトは、特に、デーモンなどの常駐型アプリケーションを起動する際によく使われます。例えばJavaサーブレットコンテナとして有名な**Tomcat**では、起動するためのスクリプトstartup.shを次のように記述してcatalina.shをラップしています。

リスト1 startup.shはcatalina.shをラップして実行する

```
...
PRGDIR=`dirname "$PRG"`
EXECUTABLE=catalina.sh
...
exec "$PRGDIR"/"$EXECUTABLE" start "$@"
```

興味のある方は、TomcatのWebページ http://tomcat.apache.org/ からダウンロードして、スタートアップスクリプトを読んでみてください。

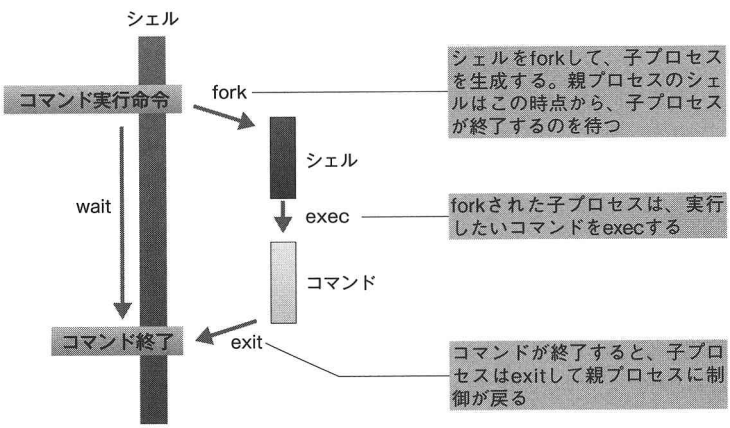
さて、ラッパースクリプトでプログラムを実行する場合は、❶のように**execコマンド**を用いるのが一般的な手法です。単に外部プログラムを起動するだけなら、次のように書いても問題はありません。しかしこれは、常駐型のサーバプログラムを起動する際には、あまり使われない書き方です。

```
./myappd "$@"
```

なぜ上記のように書かずにexecコマンドを使ったほうがよいかを理解するには、シェルが外部プログラムを起動する際のOSの動きについて理解する必要があります。そこで以下に、シェルとコマンド実行の流れを簡単に説明します。

シェルがコマンドを実行する際には、内部では次のような動作をします。なお、ここであげているfork、exec、exitというのはシステムコールであって、コマンド名ではないことに注意してください。

外部プログラムを起動する際のOSの動き



この際、OSにとってもっとも負荷が高いのは、forkによるプロセス生成です。そのためプログラマは、できるだけ無駄なプロセスを生成しないようにするべきです。

❶で利用しているexecコマンドを用いると、シェルはforkを行わず、現在のシェルで直接execシステムコールを発行します。結果として、余計なプロセスを生成する必要がありません。もしここで、exceを使わずコマンドを直接実行すると、シェルはmyappdが終了するのを待ち続けます。myappdが常駐型のサーバプログラムの場合には、この親プロセスのシェルは無駄なプロセスとなってしまいます。execコマンドを利用することで、この無駄なプロセスが発生することを防いでいるのです。

シェル1つくらいならば大したことはありませんが、例えばこのmyappdがサーバ常駐

型のプログラムで同時に複数起動されるケースを考えると、複数のシェルプロセスが無駄に存在することになります。そのためこのサンプル例のように、ラッパースクリプトを書く際は、リソースの無駄使いを防ぐためexecコマンドを用いて外部プログラムを実行するのがお勧めです。

なお、❶では、"$@"と記述して、シェルスクリプトに指定されたコマンドライン引数を実際のコマンドにそのまま渡しています。**$@はコマンドライン引数全体を表す特殊な変数**で、"$@"と指定することで位置パラメータを順にクォートしたのと同じことになります。

$@でパラメータをクォートするときの動作

記法	意味
$@	$1 $2 $3 ···
"$@"	"$1" "$2" "$3" ···

こうして"$@"を実行するコマンドの引数として渡すことで、元の引数をそのまま外部コマンドに引数指定して実行できます。

注意事項

・コマンドライン引数すべてを意味する表現として、他に$*があります。しかしこの変数は、このようなラッパーの例では利用できません。

$*でパラメータをクォートするときの動作

記法	意味
$*	$1 $2 $3 ···
"$*"	"$1 $2 $3 ···"

ダブルクォートでくくった場合、位置パラメータごとに""でくくるのではなく、引数全体が1つの文字列として扱われてしまいます。そのため実用上、引数を操作する際に$*を利用することはあまりありません。基本的に"$*"は使わず"$@"を用いると覚えておいてよいでしょう。

No. 091 scpでファイル転送を行ってCPU利用率を計算し、圧縮処理をすべきかどうか判断する

利用コマンド

time, scp, awk, bc

キーワード

CPUバウンド, I/Oバウンド, 処理時間, 計測

いつ使うか

スクリプトの実行時間とCPU利用時間を計測して、全体の実行時間に対するCPU利用率を知りたいとき

実行例

```
$ ./time-scp.sh
transfer.dat          100%   18MB   9.0MB/s   00:02
scp転送時 CPU利用率: 66.98 (%)
```

スクリプト

```
#!/bin/sh

# テスト転送ファイルのファイル名、転送先などの定義
username="user1"          # sshユーザ名
filename="transfer.dat"   # 転送ファイル名
hostname="192.168.2.10"   # 転送先のホスト
path="/var/tmp"           # 転送先パス
tmpfile="timetmp.$$"      # 時間計測のための一時ファイル

# scpコマンドでファイルを転送。
# timeコマンドで時間を測り、一時ファイルに出力する
(time -p scp -C "$filename" ${username}@${hostname}:"${path}" ) 2>
"$tmpfile"

# timeコマンドの出力一時ファイルから、各timeを抽出
realtime=$(awk '/^real / {print $2}' "$tmpfile")
usertime=$(awk '/^user / {print $2}' "$tmpfile")
systime=$(awk '/^sys / {print $2}' "$tmpfile")

# CPU利用時間から、CPU利用率を計算する
cpu_percentage=$(echo "scale=2; 100 * ($usertime + $systime) / $realtime" |
bc )
echo "scp転送時 CPU利用率: $cpu_percentage (%)"

# 一時ファイルの削除
rm -f "$tmpfile"
```

❶ ❷ ❸ ❹

解説

このスクリプトは、scpコマンドでファイルを転送する際にファイル圧縮を行い、その際の全体でかかった時間に対するCPU処理時間の比率を表示するものです。なおこの例では処理時間を測るため、パスワードを手で入力する時間を含めないようにするためにサーバ間は**ssh公開鍵認証**を行い、パスワード入力は行われないものと仮定しています。

ここでssh公開鍵認証とは、**秘密鍵**と**公開鍵**の鍵ペアを用いて認証を行う方式です。一般的にsshでパスワード認証を利用する場合には、ログイン時にパスワードを入力する必要があるため、シェルスクリプトでの自動化が難しくなります。一方、鍵認証によるログイン方式を用いれば、パスワード入力が不要となるため自動化が簡単に行えます。

この鍵認証のssh設定について詳しく解説していると本書の範囲を大きく超えてしまいますので、ここでは以下で手順のみを簡単に紹介します。

1) OpenSSLのssh-keygenコマンドなどを利用して、秘密鍵・公開鍵の鍵ペアを生成する。
2) 作成した鍵ペアのうち、公開鍵ファイルの内容を、接続先ホストの~/.ssh/authorized_keysファイルに追記する。
3) 接続元ホストの~/.ssh/configファイルを編集し、秘密鍵のファイルやログインユーザ名の指定を行う。

なお、秘密鍵ファイルはとても重要なファイルです。設置したディレクトリのパーミッションは他人に読めないようにしておくなど、厳重に管理してください。

このサンプル例でファイル転送に用いている**scpコマンド**は、**-Cオプション**を付けることでファイルを圧縮して送信することができます。しかしネットワーク帯域は十分にあるのにCPUが貧弱なマシンでこの圧縮処理を行うと、逆にCPU時間を食って全体の転送時間が長くかかってしまいます。一方、ネットワークが極端に遅い環境では、圧縮したほうが効率的に転送できるでしょう。このスクリプトは、このような状態を判断するツールとしての利用を想定しています。

❶で、まず転送に用いるファイルやサーバ名などの設定を行っています。シェル変数tmpfileは、次のtimeコマンドの出力を保存する一時ファイルです。

❷では、scpコマンドでファイルを転送しています。scpコマンドは、次のような書式で通信先を指定してファイルを転送することができます。❷ではCPU時間を計測するため、圧縮するための-Cオプションも利用しています。

書 式 scpコマンドの書式

・ファイル圧縮して転送する場合

```
scp -C <ファイル名> <ユーザ名>@<サーバ名>:<転送先パス>
```

・ファイル圧縮せず、そのまま転送する場合

```
scp <ファイル名> <ユーザ名>@<サーバ名>:<転送先パス>
```

また、❷で利用している**timeコマンド**は、指定したコマンドの実行時間とCPU利用時間を計測するコマンドです。例えばcpコマンドでファイルコピーするのにかかった時間を、timeコマンドで計測する場合は、次のようにします。**-pオプション**は、単純に秒数のみを表示したい場合に使用します。

timeコマンドで実行時間を計測

```
$ time -p cp tmp.dat tmp2.dat
real 3.92
user 0.02
sys 0.33
```

ここで、realはコマンドの起動から終了までに経過した実時間、userはユーザCPU時間、sysはシステムCPU時間を指します。つまり大ざっぱに見れば、user + sysがCPU時間、そしてrealからCPU時間を引いた時間がI/O待ちの時間です。

CPU処理した時間：user + sys
I/O待ち時間　　：real - (user + sys)

一般に、CPU処理の時間が長い場合には「CPUバウンドな処理」、ディスクやネットワークなどのI/O待ちが長い場合には「I/Oバウンドな処理」と呼びます。このサンプル例ではtimeコマンドの出力結果を利用して、CPU利用時間の率を計算しています。これにより、CPUバウンドな処理なのかI/Oバウンドな処理なのかを判断できるでしょう。

さて、リストの❷に戻ります。❷では、コマンド全体を丸カッコ()でくくって**サブシェル**としています。これはコマンド全体をグルーピングしてリダイレクトするための処理です。サブシェルを使わずに書くと、scpコマンドの結果のみがリダイレクトされてしまい、timeコマンドの結果が得られません。またtimeコマンドは結果を標準エラー出力に表示するため、❷では標準エラー出力を「2>」としてファイルにリダイレクトしています。

ここまででscp転送時のCPU利用時間と実時間を測定して、一時ファイル$tmpfileに出力できました。次にCPU処理時間の率を計算してみましょう。

❸では、一時ファイルからtimeコマンドの出力real, user, sysの値を取得しています。**awkコマンド**でそれぞれのパラメータをフィルタ指定し、第2カラムの値を取得してシェル変数にそれぞれ代入しています。これで転送時間全体に対する、CPU利用時間の率が計算できます。この計算をしているのが❹です。シェルスクリプトでの数値計算にはexprコマンドがよく用いられますが、**exprコマンドは小数を含む計算ができません**。小数を含む計算はbcコマンドを利用します。

bcコマンドは、次のように計算式を標準入力に与えることで数値計算ができるコマンドです。小数点以下の桁数は、scale=で指定します。

小数点第2位まで出力するbcコマンドでの計算例

```
$ echo "scale=2; (32.2 + 41.3) / 5.6" | bc
13.12
```

前述のとおり、CPU処理した時間はuser+sysですから、これを全体でかかった時間realで割れば、CPU処理時間の率を見積もることができます。❹ではこの値を計算し、パーセント表示しています。

もしこのサンプル例で表示されたCPU処理時間のパーセンテージが非常に大きい場合、scpコマンドでの転送時に、むしろ圧縮しない(-Cオプションを付けない)ほうが転送は早く終わるかもしれません。ネットワーク帯域やCPU性能によって判断は変わるでしょうから、その指標のツールとして利用してみてください。

注意事項

・timeコマンドはシェルビルトインのコマンドと外部コマンドがあり、それぞれ環境によって出力形式が変わります。例えばUbuntuの場合、以下3つのtimeコマンドがあります。

▶shビルトインのtimeコマンド
▶bashビルトインのtimeコマンド
▶外部コマンド /usr/bin/time

シェルのビルトインコマンドでも外部コマンドでも、timeコマンドで-pオプションは利用できます。表示形式の環境依存をなくすため、シェルスクリプトでは基本的に-pオプションを利用するとよいでしょう。

関連項目

065 指定したサイズのファイルを作り、転送速度を測定する

No. 092 移植性を考慮して外部コマンドを利用する

利用コマンド
uname, echo

キーワード
汎用性, 移植性, OS

いつ使うか
OSごとにコマンドの動作が異なる場合に、それぞれのコマンドを変数として定義したいとき

実行例

```
$ ./os-command.sh
ここは改行をしないメッセージです。
```

スクリプト

```
#!/bin/sh

# echoコマンドのパスを、環境によって変えてシェル変数ECHOに
# 代入する。
case $(uname -s) in ―――――――――――――❶
  # Macの場合はシェルビルトインではなく/bin/echoを用いる
  Darwin)
    ECHO="/bin/echo"  ―――――――――――❷
    ;;
  *)
    ECHO="echo"
    ;;
esac

$ECHO -n "ここは改行をしない"  ―――――❸
$ECHO "メッセージです。"
```

解説

このスクリプトは、echoコマンドをシェル変数ECHOで置き換えるものです。**Macのechoコマンドは改行させない-nオプションに対応していない**ため、この対策のためのサンプル例です。

P.15で紹介したように、Macのシェルビルトインのechoコマンドは、改行させない-nオプションに対応していません。そのため、Linuxなどからスクリプトを移植する際には、echo -nしている箇所をそのつど修正しないといけません。そのような手間を嫌って、できるだけ汎用性を持たせようとしたのがこのサンプル例となります。

シェルスクリプトで、あるコマンドを変数名で置き換える際には、そのコマンドを大文字とした変数名がよく使われます。これは慣習的なものですから、特に技術的な理由があるわけではありません。しかし他の人が書いたスクリプトを読む際にこのような表現がでてくることもあるので、知識として覚えておいたほうがよいでしょう。このサンプル例ではechoコマンドを置き換えるため、シェル変数ECHOを用いました。

まずOSを判定するために、❶ではcase文の引数としてunameコマンドを実行しています。**unameコマンド**はシステム情報を表示するコマンドで、OSの名称を表示するには**-sオプション**を利用します。実際に得られる文字列を次の表にまとめました。

OSごとのunameコマンドの表示

OS	uname -s の出力
Linux	Linux
FreeBSD	FreeBSD
MacOS	Darwin
Solaris	SunOS

❷では、Macの場合(Darwinの場合)には、シェル変数ECHOに**/bin/echo**と指定して、シェルビルトインのechoではなく外部コマンドの/bin/echoを用いるようにしています。それ以外のOSでは、そのままechoコマンドを実行したいので、シェル変数ECHOにもechoと入力しています。

こうして環境ごとのechoコマンドが準備できましたので、❸で実行しています。このようにコマンドを変数で置き換えて、新しく定義した変数をコマンドのようにみなしてスクリプトを書きます。ここでは1行目が-n付きで実行されていますので、改行されません。実行例のとおり、次のechoコマンドとあわせて1行で表示されます。このようにして、OSごとにコマンドの挙動が若干異なる場合にも、同一のスクリプトで複数のOSに対応したスクリプトを書くことができます。

注意事項

- 汎用性を重視するとスクリプトは煩雑になります。この例で見ると、たかだかechoコマンド1つに面倒なことをしすぎだという印象を受ける読者もいるでしょう。これはそのとおりで、あまり汎用性を重視しすぎてスクリプトが複雑になっては本末転倒です。ただし、このようなOS汎用性を重視して書かれたスクリプトは、特にオープンソース製品には多いため、知識として知っておいたほうがよいでしょう。
- gzipコマンドは、環境変数GZIPがデフォルトのオプションとして使われます。そのため、このような手法には注意が必要なコマンドです。

関連項目

005 ユーザのキー入力を1文字だけ取得する(入力時にEnterを不要にする)

No. 093 リダイレクトが煩雑とならないよう、グルーピングして見通しをよくする

利用コマンド
echo

キーワード
リダイレクト, グルーピング

いつ使うか
複数のコマンドの結果を同一のファイルにリダイレクトして出力するとき、何行も繰り返し書くのをやめてすっきりさせたいとき

実行例

```
$ ./echo-redirect.sh
$ cat output.log
[Script start]
Sat Jan 18 22:21:22 JST 2014
echo-redirect.sh
output.log
script.sh
[Script end]
```

スクリプト

```
#!/bin/sh

# 中カッコでグルーピングし、リダイレクトを1つにまとめる
{ ──────────────────❶
  echo "[Script start]"
  date
  ls
  echo "[Script end]"
} > output.log ──────────❷
```

解説

このスクリプトは、複数のコマンドの結果をまとめて、1つのファイルにリダイレクトして出力するものです。{}(中カッコ)による**グルーピング**を行っているのがポイントです。実行するコマンドとして、ここではdateコマンドで日付を表示してからlsコマンドを実行しています。ここは読者が実際に使うコマンドに置き換えてみてください。

シェルスクリプトを書いていると、次のように、コマンドの結果を同一のファイルへリダイレクトする箇所が何行にも渡って続く場合があります。こういう書き方は見づらく煩雑で、何度も同じことを書くのは面倒でミスも招きます。

```
echo "[Script start]" > output.log
date >> output.log
ls >> output.log
echo "[Script end]" >> output.log
```

このようにリダイレクトが続くときは❶のように、中カッコを用いてコマンド群をグルーピングします。グルーピングを使えば、❷のように、最後に一度だけリダイレクトを書くだけですむためスクリプトが煩雑になりません。こうすれば全体が見やすくすっきり書けますし、リダイレクト先のファイルを変えたい場合も1カ所の訂正のみですむため保守性が向上し、コーディングスタイルとしても優れた書き方になります。

なおグルーピングは中カッコではなく、()(丸カッコ)で行う例もあります。丸カッコでグルーピングすると、その中のコマンドはサブシェルで実行されます。このサンプル例では特に違いはありませんが、例えばcdコマンドでディレクトリ移動をした場合には、サブシェルから抜けると元のディレクトリに戻ってしまう点が、中カッコでのグルーピングと異なる点です。このサブシェルの動作については、P.66で解説していますのでそちらを参照してください。

注意事項

・次のように、中カッコ内にコマンドを1行で並べて書くこともできます。短いコマンド2〜3個ならば、この書き方のほうがコマンドをひとまとめにしているという意味がはっきりわかって見やすいかもしれません。

```
{ echo "[Script start]"; date; ls; echo "[Script end]"; } > output.log
```

なお、1行に並べて書く際は、コマンドごとにセミコロンが必要となります。特に最後のセミコロンを忘れることが多いので注意しましょう。

関連項目

025 ディレクトリ移動した後に簡単に元の場所に戻る

No. 094 コマンドがどこかで失敗したらそこで終了し、スクリプトの誤作動を防ぐ

利用コマンド
set, cd, rm

キーワード
終了ステータス, コマンド, 失敗, 停止

いつ使うか
何かしらの重要な処理を行うスクリプトで、途中のコマンドが1つでも失敗すれば、そこでただちに停止したいとき

実行例

```
$ ./set-e.sh
./set-e.sh: line 12: cd: /var/log/myapp-: No such file or directory
$ ls
set-e.sh  test.log      (test.logが誤削除されていない)
```

スクリプト

```
#!/bin/sh

# コマンドの終了ステータスが非ゼロの場合は、
# スクリプトをただちに終了する
set -e ―――――――――――――――――――――――――❶

# 削除ファイルの格納ディレクトリ（をミスタイプしている）
deldir="/var/log/myapp-" ―――――――――――――❷

# ディレクトリ$deldirに移動して、拡張子.logのファイルを削除する。
# set -e しているため、ディレクトリ移動に失敗すればrmコマンドは実行されない
cd "$deldir" ―――――――――――――――――――――❸
rm -f *.log ―――――――――――――――――――――――❹
```

解説

このスクリプトは、シェル変数deldirで指定されたディレクトリの中の、拡張子.logというファイルをすべて削除するものです。定期的にこのスクリプトを動かして、何かのアプリケーションの古いログファイルを削除してディスクスペースを空けるなどの用途を想定しています。このサンプル例では、コマンドが失敗すればそこでスクリプトの実行を停止して、途中処理がエラーのまま先の処理に進まないようにしていることがポイントです。

シェルスクリプトでは、スクリプトファイルに書かれたとおりに、上から順にコマンドを実行していきます。この際、途中のコマンドが成功したか、失敗したかは関係ありませ

ん。つまり、1つ前のコマンドが成功しているという前提でスクリプトを書いていくと、実は1つ前のコマンドが失敗していて思わぬバグを招き入れることがあります。

この解決策が、サンプル例であげた**setコマンド**の**-eオプション**の利用です。このオプションを利用すると、途中に終了ステータスが非ゼロのコマンドがあるとそこでスクリプトが終了します。

本サンプルでは、まず❶でset -eオプションを実行しています。set -eは、一般的にはこのようにスクリプトの先頭に書くのが普通ですが、スクリプトの特定の行以降だけにチェックを入れたいという場合には、途中からset -eを書くケースもあります。一方、途中からset -eの効果を抑制するときの方法は後述します。

❷で、ログファイルを削除するディレクトリを指定しています。ここでは対象ディレクトリをタイプミスしたというケースを想定していますから、"/var/log/myapp"が本来のディレクトリなのに、後ろにゴミが付いて"/var/log/myapp-"となってしまった、という設定です。

❸で、存在しないディレクトリに移動しようとしているので、**cdコマンド**はエラーになり、終了ステータスに非ゼロを返します。このスクリプトは冒頭でset -eしているため、ここで即座にスクリプトは終了します。結果として、.logファイルを消すrmコマンド(❹)は実行されません。もしset -eしていないと、❹はcdコマンドに失敗しているのにそのまま実行され、結果としてカレントディレクトリの.logファイルが削除されてしまいます。

このようにして、コマンドが途中で失敗した場合にスクリプトをただちに停止させることができます。ファイルの削除を行うスクリプトを書く際に、誤動作を防止するために役立つでしょう。

set -eの詳細

一般にシェルスクリプトで「コマンドが成功」するということは、コマンドの終了ステータスが0であることを指します。コマンドの終了ステータスはシェルの**特殊変数$?**に入っていますので、この値を参照すれば成功・失敗が判断できます。この$?変数の扱いについては、P.25で詳しく紹介していますのでそちらを参照してください。

setコマンドの-eオプションは、コマンドの終了ステータスが非ゼロならば、何でも終了してしまいます。気を付けないといけないのは、例えばdiffコマンドで「ファイルの差分を出力」を意図して次のような処理を書いても、diffコマンドは差分があると終了ステータスに1を返すため、スクリプトが停止してしまうようなケースです。

```
diff from.txt to.txt > diff.txt
```

このように「予期したコマンド結果ではあるが、終了ステータスが非ゼロ」なコマンドを含むスクリプトでset -eを使いたい場合には、いくつか対応方法があります。ここでは2つの例を紹介します。

❖ (1)一時的に「set +e」として、set -eを無効にする

setコマンドの**+eオプション**を使えば-e指定を無効にできます。そのため、非ゼロを返すコマンドの直前で+e、直後で-e指定すれば一時的にset -eを無効化できます。

```
set +e
diff from.txt to.txt > diff.txt
set -e
```

❖ (2) || : を用いる

set -eは、パイプラインの最後のステータスのみを対象とします。そこで、前のコマンドの終了ステータスが非ゼロならば実行されるように**OR演算子(||)**を用いて後ろに**: (ヌルコマンド)**を実行しておけば、ヌルコマンドは常に成功して0を返すため、set -eに引っかかりません。このOR演算子の使い方は、P.200で解説しています。

```
diff from.txt to.txt > diff.txt || :
```

なおここで、ヌルコマンドは、代わりにtrueコマンドを用いてもかまいません。ヌルコマンドについてはP.71で詳しく紹介しています。

注意事項

- setコマンドの-eオプションを付けるとエラー時にスクリプトは停止するため、スクリプト内にはエラー処理を書かないコーディングスタイルとなります。これには賛否両論あり、「set -eは禁止として一切使わずに、自分できちんとエラー処理を書くべきだ」という主張もあります。目的や場合に応じての使い分けが必要でしょう。

関連項目

027 ファイルの中身を消去して、ゼロバイトの空ファイルにする
103 Webサーバからファイルをダウンロードして、ファイルのMD5ハッシュ値を計算する

No. 095 複数のURLからファイルを並列で同時ダウンロードする

利用コマンド
curl

キーワード
バックグラウンド, 並列処理, ダウンロード

いつ使うか
待ち時間があるコマンドが複数あるとき、並列に実行して全体の実行時間を短くしたいとき

実行例

```
$ ./background-download.sh
$                                    ファイルのダウンロード完了を待たず
                                     にすぐにシェルに制御が戻ってくる
```

スクリプト

```
#!/bin/sh

# 並列ダウンロードのため、複数のサイトからのダウンロードを
# それぞれをバックグラウンドで処理する
curl -sO http://www.example.org/download/bigfile.dat &
curl -sO http://www.example.com/files/sample.pdf &
curl -sO http://jp.example.net/images/large.jpg &
```

❶

解説

このスクリプトは、複数のWebサイトからのファイルダウンロードを並列に実行するものです。**curlコマンド**はWebサイトからファイルをダウンロードするコマンドで、ここでは途中経過を出力しない**-s (slient) オプション**と、標準出力ではなくファイルとして保存する**-Oオプション**を利用しています。

サンプル例では、curlコマンドを**バックグラウンドで動かす**ことで、ダウンロードを並列実行します。❶のように、コマンドラインの一番後ろに**&**を付けると、シェルはコマンドの終了を待たずに次の処理に移ります。そのため本サンプルでは、1つ目のファイルbigfile.datのダウンロード完了を待たずに、次のファイルsample.pdf、そして次のファイルlarge.jpgのダウンロードが同時に開始されます。

コマンドは3つともバックグラウンドで処理されていますので、結果としてスクリプトを実行すると、すぐにシェルに制御が戻ります。このようにして、効率的にファイルをダウンロードすることができます。

なお、標準エラー出力を標準出力へとまとめてリダイレクトする「2>&1」を併用する場合は、&の位置に迷うことがあるかもしれません。このようなケースでは、リスト1のようにバックグラウンド実行の&を一番後ろに付けることが正解です。

リスト1 バックグラウンド実行とリダイレクトの同時指定

```
#!/bin/sh

# 長い時間がかかるコマンドlong_time_programの表示結果を
# result.logにリダイレクトして、バックグラウンドで実行する
long_time_program > result.log 2>&1 &
```

ただし、並列してプロセスを実行すると、当然のことながらサーバに負荷がかかります。そのためたくさんの処理をバックグラウンドで同時実行させる場合には、事前に性能検証が必要でしょう。また、例えばディスクアクセスを激しく行うプログラムでは、複数のコマンドを同時に動かしても、それぞれがディスク利用待ちとなる時間が長くなり、並列して動かしても大して実行時間に変わりはないことがあります。何でもバックグラウンドで動かせばよいというわけではないので、注意しましょう。

注意事項

- 同一のWebサイトに並列して大量のダウンロードを行うことは、迷惑行為にあたり、度がすぎればDoS攻撃とみなされる可能性もあります。そのためこのサンプルは、同一Webサイトへ大量のアクセスをかける用途には用いないでください。
- このスクリプトは各コマンドをバックグラウンドで動かすため、実行と同時にシェルに制御が戻ってきます。そのためすべてのコマンドが終了したかどうかは、出力しているファイルを個別にチェックする必要があります。
- 「各コマンドは個別にバックグラウンドで動かしたいが、スクリプト自体はすべてのコマンドが終了するのを待ってから終了したい」という場合には、**waitコマンド**を利用します。P.266を参照してください。
- FreeBSDではcurlは標準でインストールされていません。そのため次のように、代わりにfetchコマンドを利用するとよいでしょう。

```
fetch "http://www.example.org/download/bigfile.dat"
```

関連項目

096 多数のホスト宛てにpingを投げる際、並列して実行し待ち時間を減らす

No. 096 多数のホスト宛てにpingを投げる際、並列して実行し待ち時間を減らす

利用コマンド
ping, wait, cat

キーワード
並列処理, 同期, 終了

いつ使うか
複数のホストに対してpingコマンドを実行する際、待ち時間を減らすために並列してバックグラウンド実行させ、さらにコマンドの終了を同期させて結果を順に出力したいとき

実行例

```
$ ./background-wait.sh
PING 192.168.2.1 (192.168.2.1): 56 data bytes
64 bytes from 192.168.2.1: icmp_seq=0 ttl=255 time=3.554 ms
64 bytes from 192.168.2.1: icmp_seq=1 ttl=255 time=3.435 ms
64 bytes from 192.168.2.1: icmp_seq=2 ttl=255 time=3.469 ms
(省略：2つ目、3つ目のping結果が順に表示される)
```

スクリプト

```
#!/bin/sh

# 3つのホストに並列してpingを実行する。6回実行するため、
#それぞれ約5秒の待ち時間がかかる
ping -c 6 192.168.2.1 > host1.log &
ping -c 6 192.168.2.2 > host2.log &
ping -c 6 192.168.2.3 > host3.log &                    ❶

# 3つのpingコマンドが終了するまで待ち、同期をとる
wait                                                   ❷

# pingコマンドの結果を表示する
cat host1.log host2.log host3.log                      ❸
```

解説

このスクリプトは、複数のサーバにpingコマンドを実行してネットワークの疎通を確認し、その結果を表示するものです。pingコマンドはバックグラウンドで並列に実行させますが、最後に結果を表示する前に**waitコマンドで同期をとっている**点がポイントです。なお、pingコマンドとICMPパケットについては、P.323で紹介していますので詳しくはそ

ちらを参照してください。

❶では、「192.168.2.1」、「192.168.2.2」、「192.168.2.3」という3つのホストに対し、**pingコマンド**でICMPパケットを送っています。この際、結果をそれぞれログファイル「host1.log」、「host2.log」、「host3.log」として出力しています。ICMPパケットを送る回数は、-cオプションを用いて6回と指定しています。

pingコマンドでは、ICMPパケットを複数回送る際には、自動的に1秒のウェイトを挿入します。つまり6回pingを実行する場合、あいだにウェイトが5回入るため、最低でも5秒はかかります。サンプル例では3台にpingを実行しているため、全体としては最低でも5秒×3台＝15秒かかってしまいます。台数が増えれば待ち時間はもっと増えてしまうでしょう。

そこで❶では時間短縮のため、3つのpingコマンドそれぞれの最後に**&**を付けて、pingコマンドをバックグラウンドで実行（→P.264）しています。このようにバックグラウンドで並列実行すれば、（応答時間を無視すれば）3つのpingコマンドは5秒ほどで終わります。

❷ではwaitコマンドを利用して、バックグラウンドで実行している3つのpingコマンドの同期をとっています。単にコマンドを並列に処理してバックグラウンド実行させるだけでよければ、&を付けて実行するだけで済みます。しかしそれではコマンドの終了を検知できないため、スクリプトの最後に結果を表示することができません。このサンプル例のように終了結果を表示したい場合には、waitコマンドで同期をとる必要があります。

waitコマンドを引数なしで実行すると、スクリプトから起動された、バックグラウンド実行されているコマンドすべての終了を待ちます。すなわちサンプル例では、3つのpingコマンドの終了を待つことになります。本サンプルでは最後にレポートとしてping結果を順番に表示したいため、すべてのpingコマンドが終了するのを待つ必要があります。そのためwaitコマンドを使うことで、それぞれバックグラウンドで実行しているpingコマンドが「待ち合わせ」て、pingコマンドの終了を待ってから次の行に進むようになります。

❸で、3つのpingコマンドの出力結果をcatコマンドで表示しています。❷でwaitコマンドを利用しましたから、このcatコマンド実行時にはpingコマンドは3つとも終了しており、結果レポートを正しく表示することができます。

注意事項

- **あまりにたくさんのバックグラウンド処理を並列させるとネットワークが混み合い、このようなネットワーク診断ツールでは異常な結果をもたらすかもしれません。そのため同時実行するコマンドの数は結果を見ながら適宜、調整する必要があります。**

関連項目

095 複数のURLからファイルを並列で同時ダウンロードする
114 サーバのping監視を行う

No. 097 シェルスクリプトの一部にPerlやRubyを使う

利用コマンド
perl, nc, sleep

キーワード
ワンライナー , Perl, Ruby, 乱数

いつ使うか
乱数を生成するなど、シェルスクリプトの機能の一部としてPerlやRubyのコードを書きたいとき

実行例

```
$ ./perl-oneline.sh
Connection to 192.168.2.1 80 port [tcp/http] succeeded!
Wait: 4 sec.
Connection to 192.168.2.1 80 port [tcp/http] succeeded!
```

スクリプト

```
#!/bin/sh

# テスト通信先の定義
ipaddr="192.168.2.1"
port=80

# 1から10までの整数値の乱数を、Perlのワンライナーで生成する
waittime=$(perl -e 'print 1 + int(rand(10))')

# テストコマンドを、ウェイトを入れて2回実行する
nc -w 5 -zv $ipaddr $port
echo "Wait: $waittime sec."
sleep $waittime
nc -w 5 -zv $ipaddr $port
```

❶ ❷ ❸

解説

このスクリプトは、**ncコマンド**によるネットワークの通信テストを2回行う際、合間のウェイト秒として乱数を用いるサンプル例です。乱数は**Perl**を用いて生成しています。

ネットワークテストの際、常に定数のウェイト値を利用すると、タイムアウトの設定値などに見落としが生じる可能性があります。そのため、このようにネットワークテストツールのウェイト秒数に乱数を入れてみるのは、ときおり使われる手法です。

まず❶で、ncコマンドでテストする対象のIPアドレスとポート番号を定義しています。ncコマンドについて詳しくは、P.162で解説していますのでそちらを参照してください。

❷では、Perlの処理結果をシェルスクリプトから利用しています。PerlやRubyなどのスクリプト言語はとても強力で、ワンライナーと呼ばれる、1行だけ書いたスクリプトもよく使われます。ここでは、シェルスクリプトからPerlのワンライナーを利用する例を紹介します。PerlやRubyの機能をシェルスクリプトから使いたいとき、何かと役に立つことがあるでしょう。

Perlでワンライナーを書くには、perlコマンドの**-eオプション**を利用して、コマンドライン引数に直接Perlスクリプトを記述します。例えば❷の行は、きちんとPerlスクリプトとして書くと、リスト1のようになります。

リスト1 乱数を発生させるPerlスクリプト

```
#!/usr/bin/perl

print 1 + int(rand(10));
```

このようなワンライナーを、❷のように**コマンド置換$()**の中に書くことにより、結果をシェルスクリプトのシェル変数として取得することができます。日付処理や数値処理、文字列処理などでPerlの力を借りると、シェルスクリプト単体では難しい処理が簡単に書けることがあります。

サンプル例では❷で、シェルスクリプトから乱数を生成するためにPerlを利用してみました。シェルスクリプトでも、/dev/urandomデバイスなどを利用して乱数生成はできますが、単純に整数値の乱数がほしいときにこの/dev/urandomを利用するのはなかなか面倒です。Perlならば**rand関数**をint型にパースすればすぐに整数の乱数は得られます。int(rand(10))では0から9の乱数を得られるため、ここでは待ち時間を作るために1を足して、1から10の乱数を得ています。

❸で、実際にncコマンドでネットワークテストを行います。1回目のテストの後に、sleepコマンドで先ほど得られた乱数値の秒数だけウェイトを入れています。このようにして、シェルスクリプトの一部に、Perlなどのスクリプト言語の処理を埋め込むことができます。

Perlによるワンライナー

サンプルで紹介した-eのほか、Perlのワンライナーでよく使われるオプションに、**-l**（出力を改行する）、**-n**（スクリプト全体が、"while (<>) { ... }" ループに囲われているものとして処理する）、があります。例えば次はsample.txtというファイルを表示するcatコマンドのような動作になります。

Perlのワンライナーを用いたファイル表示

```
$ perl -ne 'print "$_"' sample.txt
```

これをPerlスクリプトとしてきちんと書くとリスト2のような意味になり、引数で指定したファイルの中身を出力する処理を行います。

リスト2 ワンライナーの処理全体

```
#!/usr/bin/perl

while (<>) {
  print $_;
}
```

注意事項

・Rubyのワンライナーも、Perlと同様に-eオプションで記述できます。ここで利用しているto_iとは、整数型にパースするRubyのメソッドです。

Rubyの場合も-eオプションでワンライナーが可能

```
$ ruby -e 'print 1 + rand(10).to_i'
```

・何カ所にもワンライナーを埋め込むようならば、はじめからそのスクリプト言語ですべての処理を書いてしまったほうがよいでしょう。そのようなケースでは、シェルスクリプトを使うことに特別な理由がなければ、シェルスクリプト全体をPerlやRubyで書きなおすことも検討してみてください。

関連項目

061 あるサーバの特定ポートへ通信できるかのチェックスクリプト

CHAPTER

08

制御構文のサンプル

シェルスクリプトでは、if文やwhile文、for文を利用した制御構造が多用されます。本章ではこれら制御文を用いた具体的な事例として、IPアドレスのリストファイルを読み込んで順に処理する例や、無限ループでファイルをダウンロードし続けるサンプル例などを紹介します。また、前のコマンドが成功したときだけ次のコマンドを実行するAND演算子(&&)の利用例も紹介しています。

No. 098 変数を埋め込んだIPアドレスのリストファイルを読み込み、pingコマンドで疎通をチェックする

利用コマンド
sed, ping

キーワード
for文, テンプレートファイル, 置換, 変数

いつ使うか
テンプレートファイルを用いてネットワークに疎通チェックしたいとき

実行例

```
$ cat ping_target.lst
%ADDR_HEAD%.1
%ADDR_HEAD%.2
%ADDR_HEAD%.3
%ADDR_HEAD%.4
$ ./for-command.sh 192.168.2
[Success] ping -> 192.168.2.1
[Success] ping -> 192.168.2.2
[Failed] ping -> 192.168.2.3
[Success] ping -> 192.168.2.4
```

テンプレートファイル

スクリプト

```
#!/bin/sh

# コマンドライン引数をチェック
if [ -z "$1" ]; then
  echo "第3オクテットまでのIPアドレスを引数として指定してください" >&2
  exit 1
fi

# 対象のIPアドレスを、外部ファイルping_target.lstから、
# %ADDR_HEAD%の部分を置換して順に取得する
for ipaddr in $(sed "s/%ADDR_HEAD%/$1/" ping_target.lst)
do
  # pingコマンドを実行。出力結果は不要のため/dev/nullへリダイレクト
  ping -c 1 $ipaddr > /dev/null 2>&1

  # 終了ステータスで成功・失敗を表示
  if [ $? -eq 0 ]; then
    echo "[Success] ping -> $ipaddr"
  else
```

❶ ❷ ❸

```
    echo "[Failed] ping -> $ipaddr"
  fi
done
```

解説

このスクリプトは、外部ファイルping_target.lstに書かれたIPアドレスのリスト宛てに、pingコマンドで疎通を確認するものです。ここで外部ファイルは次のように、%ADDR_HEAD%という部分を実行時に置換する形式であると仮定します。例えば1行目は、%ADDR_HEAD%を「192.168.1」に置換して、「192.168.1.1」にpingを実行します。

リスト1 IPアドレスのリストが書かれたファイル(ping_target.lst)

```
%ADDR_HEAD%.1
%ADDR_HEAD%.2
%ADDR_HEAD%.3
%ADDR_HEAD%.4
```

ここではテストケースとして、「192.168.0.0/24」、「192.168.1.0/24」・・・というネットワークごとにテストする例を想定して、IPアドレスの第3オクテットまで(192.168.1など)をコマンドライン引数で指定する仕様であるとします。

このサンプル例では、for文を用いています。for文は、一般的には引数にファイルリストなどを指定して実行する例が多く見られます。例えばリスト2は、シェルのパス名展開を利用してカレントディレクトリの拡張子.htmlのファイルを順に処理しています。

リスト2 拡張子htmlのファイルに何かしらの処理をしていく

```
for filename in *.html
do
  … (何かしらの処理)
done
```

しかしfor文のinの後には、**シェルがリストと判断できるもの**であれば何を置いてもかまいません。例えばinの後ろにコマンドの実行結果を利用するために、**コマンド置換$()**を置けば、シェルは改行やスペースを区切り文字として解釈するため、実行結果ひとつひとつを順に処理することができます。つまりリスト2は、わざと丁寧に書いてみると、リスト3のようにも書くことができるわけです。

リスト3 リスト2を丁寧に書いた例

```
for filename in $(ls *.html)
do
```

```
    ... (何かしらの処理)
done
```

ただしリスト3は、ファイル名にスペースを含むファイルは正しく動作しません。厳密にリスト2と同じような動きにするならば、P.125で見たようにIFSを改行のみにする必要があります。

このように、コマンドを直接for文のinの後ろに置くやり方は、覚えておくと何かと応用が効くでしょう。

では、サンプル例を順に見ていきましょう。まず❶で、テンプレートファイルを置き換えるために指定する**コマンドライン引数**をチェックしています。1つ目の引数を意味する位置パラメータ$1を、**testコマンド**の**-z演算子**を用いて、空文字列でないかを調べています。-z演算子は空文字列ならば真を返しますので、真の場合は「引数を指定してください」と表示して終了します。

❷ではfor文の対象として、コマンド置換$()を利用して**sedコマンド**の結果を利用しています。ここでsedコマンドの対象ファイルping_target.lstは、%ADDR_HEAD%という固定文字列を持っています。これをテンプレートファイルとみなして、%ADDR_HEAD%という文字列をコマンドライン引数で指定された文字列$1で置換することで、IPアドレスを組み立ててシェル変数ipaddrに順に代入していきます。

❸で、シェル変数ipaddrで指定されたIPアドレス宛てに**pingコマンド**を実行しています。ここではpingコマンドの出力結果は不要のため/dev/nullへリダイレクトして、**終了ステータス$?**で成功か失敗かを判断し、echoコマンドで結果を表示しています(❹)。このようなpingコマンドでのネットワークテストの成功・失敗チェックについては、P.323を参照してください。

このようにして、外部のテンプレートファイルを利用した処理をfor文で実行することができます。変数を含むテンプレートファイルを作りたいときに使える手法でしょう。

関連項目

048 .svnなどの隠しファイル・ディレクトリのみを列挙する
114 サーバのping監視を行う

No. 099 連番のファイル名を持つURLを自動生成して、順にダウンロードする

利用コマンド
seq, printf, curl

キーワード
連番, URL, ダウンロード

いつ使うか
ファイル名が連番となっている画像ファイルを提供しているWebサーバから、自動的にファイル名を生成してダウンロードしたいとき

実行例

```
$ ./number-file.sh
                    URLを自動生成してダウンロードが行われる
```

スクリプト

```
#!/bin/sh

url_template="http://www.example.org/download/img_%03d.jpg" ―――❶

# seqコマンドで連番数値を生成する
for i in $(seq 10) ―――――――――――――――❷
do
  url=$(printf "$url_template" $i) ―――❸
  curl -O "$url" ―――――――――――――――❹
done
```

解説

このスクリプトは、シェル変数url_templateでURLが指定された連番ファイル名の画像ファイルを、Webサーバから順番にダウンロードするものです。ここではダウンロードしたい画像ファイルが、次のようなURLで提供されていると仮定しています。

```
http://www.example.org/download/img_001.jpg
http://www.example.org/download/img_002.jpg
http://www.example.org/download/img_003.jpg
        ⋮
```

サンプル例では、この「img_（3桁数値）.jpg」というファイル名を、for文の中でseqコマンドを使って生成しているのがポイントです。

seqコマンドは、単調増加(減少)する数値列を表示するコマンドです。利用する引数の個数によって3種類の使い方があり、

1) 最終値のみを指定する
2) 開始値と最終値を指定する
3) 開始値と差分と最終値を指定する

の3つの使い方があります。開始値と差分は、省略された場合はどちらも1とみなされます。

seqコマンドの使い方

記述例	説明	補足
seq 5	1から5まで1ずつ増加	省略された開始と差分は1
seq 3 5	3から開始して5まで1ずつ増加	省略された差分は1
seq 5 10 45	5から開始して45まで10ずつ増加	なし

seqコマンドの実行例：開始値と差分と最終値を指定する

```
$ seq 5 10 45
5
15
25
35
45
```

なお、差分にはマイナスの値を利用することもできるため、これにより減少する数列も得られます。

サンプル例では、このseqコマンドでループカウンタを作っています。このような手法には、次の2つの利点があります。

1) seqコマンドのオプションで柔軟にカウンタを作ることができる
2) ループ処理の高速化

一般に、シェルスクリプトでループカウンタに連番の数値を利用したい場合は、伝統的にはリスト1のようにwhile文の中で**exprコマンド**を用いてカウンタを増やします。

リスト1 ループカウンタの伝統的な記述法

```
i=1
while [ "$i" -le 10 ]
do
  (ここに$i を用いた処理を記述)
```

```
  # 変数iをインクリメント
  i=$(expr $i + 1)
done
```

しかしこの場合はループ1回ごとにexprコマンドを実行するため、ループ回数が多い場合には処理速度が随分と遅くなります。サンプル例のようにseqコマンドの出力をとる形にすれば、ループごとのexprコマンドによる加算が必要ないため、処理速度が速くなります。

例えばリスト2は、テスト用に1から1000までの連番ファイルを作るサンプルです。ファイルダウンロードテストなど、何かしらのテストのために、このようなファイル作成スクリプトはよく使われます。このスクリプトは筆者の環境では0.1秒もかからずに終了しました。一方、リスト1のようにループをwhile文で書いてexprコマンドでカウンタを加算すると、1.2秒ほどかかってしまいました。seqコマンドでループを作ることで、10倍以上の高速化ができたわけです。

リスト2 seqコマンドを用いてループを記述すると速度も速い

```
#!/bin/sh

for i in $(seq 1000)
do
  echo "$i" > ${i}.txt
done
```

では、サンプル例を見ていきましょう。まず❶で、ダウンロードするファイルのURLを指定します。この際に連番部分は「%03d」という、**printfコマンド**で指定する**書式指定子**で記述しています。printfコマンドと書式指定子については、P.221で詳しく説明していますのでそちらの解説を参照してください。

❷でseqコマンドの出力をfor文に指定しています。この❷の部分は、具体的に書くと次のような動作となります。こうしてシェル変数iに、順に連番の数値を代入しているわけです。

```
for i in 1 2 3 4 5 6 7 8 9 10
```

❸で、シェル変数urlに、ダウンロードする画像ファイルの実際のURLを代入しています。連番部分は「%03d」と指定しているため、例えばiが3ならば、printfコマンドは、

```
http://www.example.org/download/img_003.jpg
```

を出力します。この出力をコマンド置換$()を用いて、シェル変数urlに代入しています。

組み立てたURLを用いて、❹で**curlコマンド**でファイルをダウンロードしています。

curlコマンドには**-Oオプション**を用いて、URLのファイル名をそのまま使用してローカルに保存をしています。

このようにして、seqコマンドとfor文を組み合わせて、連番ファイル名を生成してダウンロードを行えます。この他にも、何らかのカウンタから数値を含む文字列を作るなど、連番が絡む処理にいろいろと応用できるでしょう。

注意事項

- FreeeBSDおよびMacの以前のバージョンではseqコマンドが標準でインストールされていない場合があります。代わりに、seqコマンドとほぼ似た動きをするjotコマンドがありますので、そちらを利用してください。

```
for i in $(jot 10)
```

- FreeBSDではcurlは標準でインストールされていません。そのため次のように、代わりにfetchコマンドを利用するとよいでしょう。

```
fetch "$url"
```

関連項目

081 右詰めにして数値を表示し、テキストで数値の表を作る

No. 100 強制終了されるまでファイルのダウンロードを繰り返し、通信チェックを行う

利用コマンド
true, curl, sleep

キーワード
無限ループ, ダウンロード, 接続テスト

いつ使うか
Webサーバのテスト時などに、連続したアクセスを繰り返したいとき

実行例

```
$ ./download-loop.sh
[check OK]
[check OK]
[check NG]
[check OK]
^C
```

Ctrl + C を入力するまで実行し続ける

スクリプト

```
#!/bin/sh

# チェック対象のURL
url="http://192.168.22.1/webapl/check" ―― ❶

# 無限ループを開始
while true ―― ❷
do
  # curlコマンドでテスト対象のURLからダウンロード。
  #ファイル自体は不要のため/dev/nullへ捨てる
  curl -so /dev/null "$url" ―― ❸

  # curlコマンドの終了ステータスで、OK/NGを判断
  if [ $? -eq 0 ]; then
    echo "[check OK]"
  else
    echo "[check NG]"
  fi ―― ❹

  # 5秒ウェイト
    sleep 5 ―― ❺
done
```

解説

このスクリプトは、シェル変数urlで指定されたURLから、5秒おきにダウンロードを行って結果をOK/NG表示するものです。ダウンロードは5秒間隔で行い、無限ループとなっているためキーボードから Ctrl + C を入力されるまでダウンロードを繰り返します。Webサーバの連続接続テストを行う際などに利用できるでしょう。

シェルスクリプトで**無限ループ**を記述するにはいろいろなやり方がありますが、ここでは3パターンを紹介します。どれもよく使われるため、他人の書いたスクリプトを読んで理解できるようにこの3つの書き方は覚えておいたほうがよいでしょう。

```
# 1をハードコードでの無限ループ
while [ 1 ]
do
  (処理)
done

# trueコマンドでの無限ループ
while true
do
  (処理)
done

# : (ヌルコマンド) での無限ループ
while :
do
  (処理)
done
```

中でも、「while true」の書き方は、**trueコマンド**が真を返すことは字面からも明らかなので、コードを見たときに意味もわかりやすく好まれるやり方です。

ただし、trueコマンドはシステムによっては外部コマンドであり、若干ですが機種依存性があります。最近のLinuxやMac/FreeBSDではtrueコマンドはシェルのビルトインコマンドですが、昔のSolarisなどでは外部コマンド/bin/trueでした。また、(現代のUNIXでは滅多にないことでしょうが) trueコマンドがインストールされていないシステムもあるかもしれません。そのためスクリプトの移植性を重要視する人は、trueコマンドよりも、**:(ヌルコマンド)**での無限ループ「while :」の書き方を好みます。

しかしこのサンプル例ではそのような厳密な移植性よりも、無限ループだということがひと目ですぐわかることを重視して、trueコマンドを用いました。

サンプル例では、まず❶でテスト対象のURLを指定しています。これが接続テストを行う宛先URLとなります。

続いて❷で無限ループに入ります。ここでは先述したとおり、while文とtrueコマンドの組み合わせで無限ループを作っています。

❸で、**curlコマンド**を用いて対象のURLからファイルをダウンロードしています。ここではテスト目的なので、ダウンロードしたファイル自体は必要ありません。そのため出力先を指定する**-oオプション**で**/dev/null**を指定して、ローカルにファイルは保存していません。また、curlコマンドで正しくダウンロードできたかどうかだけを知りたいため、サイレントモードを意味する-sオプションを付けて、余計な表示をしないように出力を抑制しています。

curlコマンドが正常に終了したかどうか、すなわちWebサーバから正常にファイルがダウンロードできたかどうかは、コマンドの**終了ステータス$?**が0であったかどうかで判断できます。❹で、この$?が0だったならば[check OK]、非ゼロだったならば[check NG]と表示しています。なおこの際、HTTPステータスは見ていないため、パス部をタイプミスしてWebサーバが404 Not Foundを返していたり、500 Internal Server Errorを返していても[check OK]となります。つまりこのサンプルは、あくまでWebサーバへの接続テストを行うスクリプトです。

最後に❺で、5秒間のウェイトを入れています。このような無限ループのスクリプトでは、ネットワーク帯域やCPU資源を過剰に消費してしまう恐れがあるため、このように適宜**sleepコマンド**でウェイトを入れるとよいでしょう。

注意事項

- 純粋にWebサーバへのネットワークテストのみを行いたい場合は、ファイルのダウンロード(GETメソッド)は不要ですから、コンテンツのダウンロードを行わないHEADメソッドを利用するとよいでしょう。次のようにcurlコマンドの**-Iオプション**を利用すると、HEADメソッドで接続できます。これはHTTPヘッダのみを取得します。

```
curl -I "http://192.168.22.1/"
```

- このスクリプトは無限ループのため、止めないと永久にWebサーバにアクセスし続けてしまいます。特に外部サイトにチェックを行うケースでは、離席する際には一時停止するなど、自分がすぐに操作できる範囲内でのみ動かすようにしたほうがよいでしょう。
- FreeBSDではcurlは標準でインストールされていません。そのため、代わりにfetchコマンドを利用するとよいでしょう。

```
fetch -qo /dev/null "$url"
```

No. 101 IDカラムに"00001"などゼロ詰めで書かれたCSVファイルから、番号を指定して値を抽出する

利用コマンド
read, cut, test

キーワード
数値, 文字列, ゼロ詰め

いつ使うか
"00001"などゼロ詰めされた文字列を、そのまま数値として扱いたいとき

実行例

```
$ cat data.csv
00001,Osaka
2,Kyobashi
3,Tenma
00004,Morinomiya
$ ./zero-string.sh
Osaka ―――――― ID番号が1のものが抽出された
```

スクリプト

```
#!/bin/sh

# 抽出条件などの定義
match_id=1            # 抽出するIDの値
csvfile="data.csv"    # CSVファイルを指定

# CSVファイルが存在しなければ終了
if [ ! -f "$csvfile" ]; then
  echo "CSVファイルが存在しません: $csvfile" >&2
  exit 1
fi

#CSVファイルの読み込み
while read line
do
  # 行内の各カラムをcutコマンドで取り出す
  id=$(echo $line | cut -f 1 -d ',')
  name=$(echo $line | cut -f 2 -d ',')

  # IDカラムが、シェル変数match_idで指定されたIDと一致する
  # 場合には、名前フィールドを表示する
```

❶ ❷ ❸ ❹

```
  if [ "$id" -eq "$match_id" ]; then
    echo "$name"                          ❺
  fi
done < "$csvfile"
```

解説

このスクリプトは、**CSVファイル**からID番号が1であるカラムを抽出するものです。この際、「数値が1」という表現には、"1"の他に"00001"など、頭にゼロ詰めがされている形式も扱えるようにしています。

このサンプルでは次のようなCSVファイルを扱うこととします。

リスト1 利用するCSVファイル(data.csv)

```
00001,Osaka
2,Kyobashi
3,Tenma
00004,Morinomiya
```

第1カラムにID番号、第2カラムに名前が入っています。このCSVファイルを作る前処理では、困ったことにあまりデータの整合性が考えられておらず、ID番号には先頭に0が付いているものといないものが混在しているケースを想定しています。このように、数値の頭に0が付いた数値文字列を、普通に数値として扱いたいケースが、テキスト処理の場面では多々あります。

数値文字列を純粋に数値として扱う際、実はシェルスクリプトでは難しい操作は必要ありません。真面目にやろうとすると、例えば次のように、sedコマンドなどで先頭の0を取り除くやり方が考えられます。

```
str="00001"

str=$(echo "$str" | sed "s/^0*//")
```

しかしシェルスクリプトでは、上記のような面倒な処理は必要ありません。というのも、testコマンドの同値かどうかを判定する-eq演算子や、exprコマンドなど、数値を扱うコマンドは、ゼロ詰めされた数値文字列そのままで純粋に数値として扱ってくれるからです。そのため先の例のような丁寧な置き換えは必要ないのです。

exprコマンドは先頭に0が入っていても問題なく動作する

```
$ expr "00001" + 1
2
```

これを踏まえて、サンプル例を見ていきましょう。まず❶で、抽出するID番号と、対象のCSVファイルの定義を行っています。❷では対象のCSVファイルの存在をチェックしています。**-f**は対象が通常ファイルかどうかをチェックする演算子です。それを否定演算子!と併用することで、対象がディレクトリであったり、ファイルが存在しない場合にはエラーを表示して終了するようにしています。これらファイルテストの演算子については、P.110で紹介していますのでそちらを参照してください。

❸で、CSVファイルをwhile文に入力リダイレクトして1行ずつシェル変数lineに読み込み、❹の**cutコマンド**でカラムごとの値をシェル変数id、nameにそれぞれ代入しています。このようにCSVファイルからcutコマンドで特定のカラムを取り出すやり方は、P.195で説明していますのでそちらを参照してください。

❺で、ID番号のカラムの値が、はじめにシェル変数match_idで定義した抽出すべきID番号と一致するかを判断しています。ここでは単純に、取り出した1カラム目（シェル変数id）を**testコマンド**の**-eq演算子**で確認しています。testコマンドの-eq演算子は、次のように先頭の0は気にする必要がありません。

```
# 数値として比較すれば、頭の0は気にする必要がない
[ "00001" -eq 1 ]          → True
[ "1" -eq 1 ]              → True

# 文字列として比較すると、頭の0の数が違うと偽となる
[ "00001" = "1" ]          → False
```

ただし、testコマンドで誤って=で比較すると文字列として比較されてしまうため、"00001"と"1"の比較が偽となることには注意が必要です。数値は-eq演算子で比較することに注意してください。

❺では、こうして比較して真だった場合に、抽出すべきID番号のname値を**echoコマンド**で出力しています。このようにして、数値文字列をシェルスクリプトでは簡単に扱うことができます。特にCSVファイルを扱う際に、知っておくと便利な用例でしょう。

注意事項

・値自身に,（カンマ）を含むCSVファイルは、このスクリプトでは対応していません。

関連項目

042 処理開始前に、実行権限をチェックして正常動作できることを確認してから実行する
072 CSVファイルから、指定した特定レコードのカラムの値を得る

No. 102 スクリプトを修正してif文の中身が空っぽになった際、エラーとならないようにする

利用コマンド
:（ヌルコマンド）

キーワード
if文, ヌルコマンド, 空行

いつ使うか
仕様変更などで不要となった処理をコメントアウトして、if文の中身が空になったときに、エラーとなるのを回避したいとき

実行例

```
$ ./if-null.sh
$ ―――――――――――― 何もしないで正常に終了する
```

スクリプト

```
#!/bin/sh

# データファイルの定義
datafile="/home/user1/myapp/sample.dat" ―――――― ❶

# データファイルの存在チェック
if [ -f "$datafile" ]; then ―――――――――――― ❷
  # 仕様変更で不要となったためコメントアウトしたものとする
  # ./myapp "$datafile" ―――――――――――― ❸

  # 空のif文が書けないため、:（ヌルコマンド）を置いておく
  : ―――――――――――― ❹
else
  echo "データファイルが存在しません: $1" >&2
  exit 1
fi
```

解説

このスクリプトは、シェル変数datafileで指定されたデータファイルの存在をチェックして、ファイルがあればカレントディレクトリのmyappというプログラムを実行するものです。ここでは仕様変更により、このmyappを実行する必要がなくなったためコメントアウトした結果、if文の中身が空になりエラーが出るようになってしまったというケースでの対処を想定しています。

入門書などで言及されることが少ないためあまり知られていないことですが、シェルス

クリプトでは、**if文の中身が空の場合はエラー**となります。例えばリスト1は、lock.tmpというロックファイルがあれば何もせず、なければファイルをtouchコマンドで作成するというスクリプトを書いたつもりです。

リスト1 lock.tmpがなければ作るスクリプトを書いたつもり

```
#!/bin/sh

if [ -f lock.tmp ]; then
  # ここでは何もしない
else
  # ロックファイルを作成
  touch lock.tmp
fi
```

しかし、実行すると次のようにエラーとなります。これはif文の中身が空のためです。

空のif文ではエラー

```
$ ./if.sh
./if.sh: line 5: syntax error near unexpected token `else'
./if.sh: line 5: `else'
```

C言語など一般的なプログラミング言語では、if文の中身を空っぽにしていても何も問題はありません。しかしシェルスクリプトでは、if文の中身を空にするとエラーとなります。そのため、if文の中には何か最低1つはコマンドを記述しなくてはいけません。

これはシェルスクリプト初心者が陥りがちなミスです。特に、いままで動かしていたスクリプトが仕様変更によって特定の処理が必要なくなった際に、スクリプトを丸ごと書き換えずに該当の処理だけコメントアウトして済ませるケースは多いでしょう。このような際に、コメントアウトによってif文の中で実行するコマンドがなくなると、突然エラーとなるため注意が必要です。

本サンプルは、仕様変更によってif文の中身を実行する必要がなくなった場合を想定しています。まず❶で、チェックするデータファイルの定義を行って、シェル変数datafileにファイルパスを代入しています。

❷で、対象のデータファイルの存在をチェックしています。-fは対象が通常ファイルかどうかをチェックする演算子です。このファイルテストの演算子については、P.110で紹介していますのでそちらを参照してください。

❸は、過去に外部コマンドmyappを実行していた部分です。いまは仕様変更により実行が不要となったという想定で、コメントアウトしています。つまりこのif文は、以前は次のように書かれていました。

```
if [ -f "$datafile" ]; then
  ./myapp "$datafile"
else
  echo "データファイルが存在しません: $1" >&2
  exit 1
fi
```

ここで./myappをコメントアウトすると、if文の中身が空になるためこのスクリプトがエラーとなってしまいます。これを防ぐためには、「何もしないコマンド」というものがほしくなります。このような用途のため、シェルビルトインの**:（ヌルコマンド）**が用意されています。

ヌルコマンドは、実行しても何も出力しません。実行後は必ず成功して終了ステータス0を返します。この機能を利用して、ヌルコマンドはファイルを初期化する用途に使われることもあります。ファイル初期化の用法については、P.71を参照してください。

❹で、if文の中身にこのヌルコマンドを置いています。ヌルコマンドは何もしませんから、こうして後付けで記述しても副作用がありません。そしてコマンドが実行されますから、if文の中身が空になりエラーが発生することを防ぐことができます。こうして、実質的に空のif文を書くことができます。

これは地味な例ですが、if文の中身を後からコメントアウトして空にしたためにエラーでハマってしまうというのは、運用現場でよく見られる失敗例です。そのためここで一例として取り上げてみました。

注意事項

- FreeBSDのshでは、LinuxやMacと違ってif文の中身が空でもエラーにはならないようです。しかし移植性を考慮すると、FreeBSDでも空のif文を書くのは避けて、この例のようにヌルコマンドを置いたほうがよいでしょう。

関連項目

027 ファイルの中身を消去して、ゼロバイトの空ファイルにする
042 処理開始前に、実行権限をチェックして正常動作できることを確認してから実行する

No. 103 Webサーバからファイルをダウンロードして、ファイルのMD5ハッシュ値を計算する

利用コマンド
curl, md5sum

キーワード
ダウンロード, 終了ステータス

いつ使うか
ネットワーク経由でファイルを取得する際、コピーが成功したことを確認してから次のコマンドを実行したいとき

実行例

```
$ ./andlist.sh
83036ec1109bf9770fc2d8673b545d35  sample.dat
```

スクリプト

```
#!/bin/sh

# ダウンロードするファイルのURLパス部分、ファイル名を定義
url_path="http://www.example.org/"
filename="sample.dat"                                          ❶

# ファイルをダウンロード。ダウンロード成功ならばmd5ハッシュ値を表示する。
# Mac/FreeBSDでは、md5sumコマンドではなくmd5コマンドを使う
curl -sO "${url_path}${filename}" && md5sum "$filename"        ❷

# ダウンロードファイルを削除して終了する
rm -f "$filename"                                              ❸
```

解説

このスクリプトは、シェル変数url_pathとfilenameで指定されたWebページからファイルをダウンロードして、その**MD5ハッシュ値**を表示するものです。日々変わるコンテンツの更新を定期的にチェックする用途を想定しています。ここでMD5ハッシュ値については、P.192で詳しく説明していますのでそちらを参照してください。

外部からファイルをダウンロードする際には、ネットワークの不調などで失敗する可能性が常にあることを忘れてはいけません。このサンプル例では、ダウンロードに失敗した場合にはmd5コマンドを実行しないようにして、誤ったハッシュ値を出力しないようにしています。このために、❷でAND演算子(&&)を利用しています。

AND演算子(&&)は左右に2つのコマンドを記述し、「a && b」の形で利用します。意

味としては、「a && b」は、「aとbが共に真ならば真」を意味します。しかし実用例としては、「左側のコマンドaが成功(終了ステータスが0)の場合のみ、右側のコマンドbを実行する」として使われるほうが一般的です。これは、「a && b」と書いたとき、もし左側が偽ならば全体の評価式は必ず偽になるため、右側を評価する必要がなく、左側のコマンドの終了時点で評価が打ち切りとなるからです。

つまりAND演算子は、単純に「**前のコマンドが成功したら次のコマンドが実行される**」記法である、と考えるとわかりやすいでしょう。AND演算子のこのようなロジックは、シェルスクリプトに限らず、JavaやPerlなど近年のメジャーな他のプログラミング言語でもよく出てきますから、覚えておくとよいでしょう。

サンプル例では、まず❶で、ダウンロードするファイルのURLを定義しています。ここでは次の処理で使いやすいように、URLのパス部分(url_path)とファイル名部分(filename)を別変数として定義しています。

❷で、**curlコマンド**でファイルをダウンロードしています。ここでcurlコマンドには、途中経過を表示しない**-sオプション**(silentモード)と、ダウンロードしたファイルを標準出力に表示するのではなくファイルとして保存する**-Oオプション**を利用しています。

❷では、AND演算子でcurlコマンドと**md5sumコマンド**をつないでいます。もしネットワーク不通などでcurlコマンドが失敗したときには、curlコマンドの終了ステータスは非ゼロを返します。つまり、curlコマンドが失敗した際は右側のmd5コマンドは実行されません。こうして、ダウンロードに失敗しているのにMD5ハッシュを計算してしまうような誤処理をしないで済みます。

このままではダウンロードしたファイルが残っていますので、❸で消去しています。rmコマンドには-fオプションを付けて、もしダウンロードに失敗していてファイルがない場合でも、エラーとならないようにしています。

AND演算子の詳細

AND演算子の使用例をもう少し詳しく見てみましょう。他に&&がよく使われる例としてよくあるのが、カレントディレクトリを移動するcdコマンドとセットで用いるケースです。次のような形で使われます。

```
cd "$appdir" && ./script.sh
```

シェルスクリプトは、本来ならばどこのパスから実行されても動作するように書くのが理想です。しかし実際には、カレントディレクトリにcdコマンドで移動して実行しないと、正常に動作しないものも多いでしょう。そのため、最初にcdコマンドでスクリプトの置かれたディレクトリに移動することが多くあります。

上記は、cdコマンドと、「./script.sh」という実行命令で、AND演算子(&&)を挟んでいます。こうすると、ディレクトリ名のタイプミスなどでcdコマンドによるディレクトリ移動に失敗した場合は、次の「./script.sh」は実行されません。一方、AND演算子を使わずに次のように書くと、もしディレクトリ名をタイプミスして、カレントディレクトリ

に同名のscript.shというスクリプトファイルがある場合、cdコマンドに失敗しているのに次の「./script.sh」を実行してしまいます。

```
cd "$appdir"
./script.sh
```

このようなエラーに配慮した書き方を、できる限り心がけたほうがよいでしょう。

注意事項

- curlコマンドは、HTTPステータスが404 Not Foundでも、コンテンツがダウンロードできれば終了ステータスは0を返して正常終了します。つまりこのサンプル例は、例えば404 Not Foundページのコンテンツも対象としており、HTTPステータスは無視する仕様です。
- AND演算子と対になるものとして、**OR演算子**(||)があります。「a || b」は、「aまたはbが真ならば真」を意味します。しかし実用例としては、「左側のコマンドaが失敗(終了ステータスが非ゼロ)の場合のみ、右側のコマンドbを実行する」として使われることが一般的です。例えば、次の例ではシェル変数filenameで指定されたファイルが存在するかどうかを、testコマンドで確認しています。ファイルがなければexitコマンドでスクリプトを終了します。

```
test -f "$filename" || exit 1
```

関連項目

071 入力ファイルのハッシュ値を、行ごとに追加カラムとして出力する

CHAPTER

09

サーバ管理

サーバの構築には手順書を作ることが多いのですが、必要なコマンドをシェルスクリプトで記述してしまえば、構築作業を自動化することができます。また、監視や各種チェックなどサーバの運用業務においては、手順書通りに手で実行するのではなくシェルスクリプトを提供すれば、誤ったコマンドを実行するなどのミスを無くすことができます。本章では、これらサーバ管理に必要なスクリプト例を紹介します。

No. 104 サーバのネットワークインタフェースとそのIPアドレス一覧を取得する

利用コマンド
ifconfig, awk

キーワード
NIC, IPアドレス, ネットワークインタフェース

いつ使うか
サーバのNICとIPアドレスの一覧を表示したいとき

実行例

```
$ ./nic-ipaddr.sh
[eth0]
10.211.55.18
[eth1]
[lo]
127.0.0.1
```

eth0とloのIPアドレスを表示した。eth1にはIPアドレスが振られていない

スクリプト

```
#!/bin/sh

# ifconfigコマンドで有効なインタフェースを表示して、
# awkコマンドでインタフェース名とIPアドレスを抽出する
LANG=C /sbin/ifconfig |¥                                    ❶
awk '/^[a-z]/ {print "[" $1 "]"}                            ❷
/inet / {split($2,arr,":"); print arr[2]}'                  ❸
```

解説

このスクリプトは、サーバのネットワークインタフェースと、そこに割り振られているIPアドレスを表示するものです。IPアドレス取得にはifconfigコマンドを使い、その出力をawkコマンドで加工しています。なおここでは、このサーバにはネットワークインタフェースとしてeth0とeth1があり、共にインタフェースは有効(UP)にしていますが、IPアドレスはeth0のみに設定しているという仮定をしています。

サンプルで用いている**ifconfigコマンド**は、サーバのネットワークの設定や情報取得を行うコマンドです。次のように引数なしで実行すると、サーバ上で現在有効なネットワークインタフェースの情報(リンク状態、MACアドレス、IPアドレス、転送パケット数など)を表示します。なおifconfigコマンドは、LinuxとMac/FreeBSDでは微妙に出力が異なります。そのためMac/FreeBSDをお使いの場合はサンプル例がそのまま使えませんので、「注意事項」をあわせて参照してください。

ifconfigコマンドを引数なしで実行

```
$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:1C:42:4B:9B:8B
          inet addr:10.211.55.18  Bcast:10.211.55.255  Mask:255.255.255.0
          inet6 addr: fe80::21c:42ff:fe4b:9b8b/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:53 errors:0 dropped:0 overruns:0 frame:0
          TX packets:48 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:7153 (6.9 KiB)  TX bytes:6605 (6.4 KiB)

eth1      Link encap:Ethernet  HWaddr 00:1C:42:5E:13:47
          :(省略)
lo        Link encap:Local Loopback
          :(省略)
```

ifconfigコマンドはサーバ管理でよく使われるポピュラーなコマンドですが、表示される内容が盛りだくさんなため、出力を扱うのは少々厄介です。ネットワークの設定確認などのシーンでは、このサンプル例のように、シンプルにサーバのネットワークインタフェースとIPアドレスだけ表示したいということもよくあるでしょう。

サンプルでは、❶でifconfigコマンドを実行しています。この際に、Ubuntuなどでは結果が日本語表記されてしまいIPアドレスを取り出すことが難しくなるため、英語表記で統一して表示するように先頭でLANG=Cと指定しています。また、Linux環境では/sbinにパスが通っていないことがあるため、ここではifconfigコマンドをフルパスで指定して実行しています。出力結果をパイプでawkコマンドに渡していますが、長くなるため行末に¥を置いて改行しています。

❷の**awkコマンド**では、まず/^[a-z]/というパターンを利用しています。これはインタフェース名を出力するためのパターンです。ifconfigコマンドでは、はじめに行頭にインタフェース名が表示され、その後にインデントされてリンク状態などの各種情報が表示されます。そのため行頭に小文字アルファベットがくる行をマッチし、その1カラム目を表示することでインタフェース名が得られます。ここではインタフェース名を見やすくするために、awkコマンドで表示する際、次のように[]でくくっています。

```
{print "[" $1 "]"}
```

awkコマンドではスペースは詰めて表示されますから、上記のように書くと[eth0]のようにインタフェース名をカッコ付きで表示できます。

❸で、IPアドレスを取得しています。ここで取得したいのは、「inet addr:」の後ろの部分、ここでは「10.211.55.18」です。

```
inet addr:10.211.55.18  Bcast:10.211.55.255  Mask:255.255.255.0
```

❸では、まずawkコマンドで2カラム目を$2として取り出します。awkコマンドの区切りはスペースですから、ここで$2には「addr:10.211.55.18」が入っています。

続いてこの$2からIPアドレスを取り出すには、「:(コロン)を区切り文字としてその2カラム目を取り出す」とすればよいことになります。そのため❸ではawkコマンドのsplit関数を用いて、:(コロン)で文字列を分割してarr[2]として2カラム目を取り出しています。この例のように、はじめにスペース区切りで文字を取り出し、そこからさらに別の区切り文字で取り出す場合、このようにawkコマンドの**split関数**を用いると便利でしょう。

このサンプル例ではネットワークインタフェース名とIPアドレスを出力しましたが、出力結果を別のスクリプトに渡して処理したい場合には、ネットワークインタフェース名は不要で単純にIPアドレスリストだけがほしい場合も多いでしょう。その場合、/^[a-z]/フィルタ部分を削除して次のようなスクリプトにします。

```
LANG=C /sbin/ifconfig | awk '/inet / {split($2,arr,":"); print arr[2]}'
```

注意事項

- このスクリプトでは、IPv4アドレスのみを扱っており、IPv6アドレスは無視しています。
- Mac/FreeBSDのifconfigコマンドは、Linuxのifconfigコマンドと若干出力が異なります。具体的には、IPアドレス表示部分が「inet 10.211.55.21」となり、Linuxと違って「addr:」の文字列がありません。

Mac/FreeBSDのifconfigコマンドの場合

```
$ ifconfig
em0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu
1500
    options=9b<RXCSUM,TXCSUM,VLAN_MTU,VLAN_HWTAGGING,VLAN_HWCSUM>
    ether 00:1c:42:5e:c3:b1
    inet 10.211.55.21 netmask 0xffffff00 broadcast 10.211.55.255
        :(省略)
```

そのためMac/FreeBSDの場合は:(コロン)区切りで取り出す必要がないため、次のように、Linuxよりもシンプルなスクリプトで記述することができます。

```
LANG=C /sbin/ifconfig | awk '/^[a-z]/ {print "[" $1 "]"} /inet / {print
$2}'
```

No. 105 サーバに作成済みのユーザアカウント一覧を取得する

利用コマンド
grep, cut

キーワード
ユーザアカウント, 区切り文字, カラム

いつ使うか
テキストファイルから区切り文字を指定して、特定のカラムを取り出したいとき

実行例

```
$ ./sep-cut.sh
root
bin
daemon
      :(省略)
sshd
nginx
user1
```

スクリプト

```
#!/bin/sh

# ユーザアカウント情報の対象ファイル
filename="/etc/passwd"

# 行頭が#であるコメント行を除外して、cutコマンドで、
#  * 1番目の値を表示 [-f 1]
#  * 区切り記号は : [-d ":"]
# として表示する
grep -v "^#" "$filename" | cut -f 1 -d ":"
```

解説

このスクリプトは、UNIX環境においてシステムのユーザアカウント情報が記載されたファイルである/etc/passwdファイルから、システムに存在するユーザアカウント一覧を表示するものです。サーバを構築・運用する際に、どのようなユーザが作成済みかを確認するために役立つことでしょう。

/etc/passwdファイルはユーザ管理に用いられるシステムファイルで、次のような形式となっています。

リスト1 /etc/passwdファイルの例

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
        :(省略)
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
nginx:x:498:499:Nginx web server:/var/lib/nginx:/sbin/nologin
user1:x:501:501::/home/user1:/bin/bash
```

この/etc/passwdファイルの内容はコロン区切りで記述されており、各項目の値は次のようになっています。

◆各カラムの意味

カラム	説明
カラム1	ユーザ名
カラム2	パスワード
カラム3	UID (ユーザID)
カラム4	GID (グループID)
カラム5	コメント (フルネームを入れることもある)
カラム6	ホームディレクトリのパス
カラム7	ログインシェル

ユーザアカウントやUIDを取得するために、この/etc/passwdファイルを扱うコマンドは多くあります。そこでこのサンプルでは、/etc/passwdファイルのカラム1から、ユーザ一覧を取得しています。

さてリスト1で見たように、/etc/passwdファイルの区切りは:(コロン)となっています。このようなテキストファイルから指定したカラムの値を取り出したいときは、「区切り文字を指定して」「第n番目の値を取り出す」というコマンドで対応できます。いくつか手法がありますが、サンプル例ではcutコマンドを用いています。

cutコマンドは、テキストファイルからある一部分を切り出すコマンドです。**-fオプション**を利用して、第n番目の値のみを表示することができます。なおcutコマンドの使い方はP.196で解説していますので、そちらも参照してください。また、cutコマンドでは、デフォルトの区切り記号はタブが指定されているため、❶では、区切り文字の変更を行う**-dオプション**を同時に利用しています。「-d ":"」と指定することで、:(コロン)を区切りとして指定することができるので、これで-f 1として第1番目の値、すなわちユーザ名を抽出できます。

なお、/etc/passwdファイルは、FreeBSDでは行頭に#があるコメント行が含まれています。そのためcutコマンドに渡す前に、**grepコマンド**のマッチしない行を表示するための**-vオプション**を用いて、行頭に#がある行(^#)を除外しています。

/etc/passwdファイルとパスワード

現代のUNIXでは、セキュリティの観点から/etc/passwdファイルにはパスワードは書かれておらず、代わりにLinuxでは/etc/shadowに、FreeBSDでは/etc/master.passwdに、暗号化されたパスワードが書かれています。これらのファイルはrootでしか読めませんから、rootユーザとなってファイルの中身を見てみると、次のようになっています。

暗号化されたパスワードが書かれている

```
# cat /etc/shadow
root:$6$OBO7CUwM$8hKvfUcySRdXKu6.aFF0qlLAxUL6WTPkvg6myrcl.RKdxfPlfYbr3SEs8
lk2llJrYpruxhiRTirfbMTVRz0oB0:16032:0:99999:7:::
bin:*:15628:0:99999:7:::
daemon:*:15628:0:99999:7:::
…（省略）
```

ここでは、例えば6から始まる長い文字列が暗号化されたパスワードです。昔のUNIX環境では/etc/passwdファイルにこの値が直接書かれていたのですが、悪意のあるユーザが別ユーザのパスワードを総当たり攻撃で取得することが可能となるため、現在のパスワードファイルはrootしか読めないような仕組みとなっています。

なお、Macでは/etc/passwdファイルを直接ユーザ管理には用いていないため、/etc/master.passwdファイルは存在しますが、暗号化されたパスワード自体は書かれていません。

注意事項

- Macでも/etc/passwdファイルは存在しますが、通常のログインユーザはディレクトリサービスを用いて別に管理されているため、この/etc/passwdファイルには通常ログインに用いるアカウントは存在しません。しかしその他のシステムアカウントは/etc/passwdに存在しますので、サンプル例を試してみることは可能です。
- コロン区切りのテキストファイルから一部のカラムを取り出すには、**awkコマンド**の区切り文字を指定する**-Fオプション**を利用して次のようにも書けます。

```
awk -F: '{print $1}'
```

しかしここでは、コンパクトでわかりやすいcutコマンドを用いる例を紹介しました。

関連項目

072 CSVファイルから、指定した特定レコードのカラムの値を得る

No. 106 許可したユーザのみスクリプトを実行可能とする

利用コマンド
id, whoami, echo

キーワード
実行ユーザ, パーミッション, root

いつ使うか
あるスクリプトを実行する際、指定したユーザ以外での実行を禁止したいとき

実行例

```
$ whoami
user1
$ ./id-script.sh
[ERROR] batch1 ユーザで実行してください
```

スクリプト

```
#!/bin/sh

# このスクリプトの実行を許可するユーザの定義
script_user="batch1" ―――――――――――――――――――――――――❶

# idコマンドで現在のユーザを取得し、定義と一致するかを確認
if [ $(id -nu) = "$script_user" ]; then ―――――――――――――❷
  # 許可ユーザならばバッチ処理を実行
  ./batch_program ―――――――――――――――――――――――――❸
else
  echo "[ERROR] $script_user ユーザで実行してください" >&2 ―――❹
  exit 1
fi
```

解説

このスクリプトは、現在スクリプトを実行しているユーザ名をチェックして、指定したユーザでしか実行できないように制限するものです。

バッチ処理を行うプログラムがログファイルや一時ファイルを出力する場合には、実行するユーザに注意する必要があります。スクリプトが誰でも実行できる場所に置いてあると、本来想定していたユーザとは違うユーザが実行するかもしれません。例えばこのサンプル例では、batch1というユーザが実行することを想定していますが、誤ってuser1というユーザが実行すると、(スクリプトの作りにもよりますが)パーミッションの問題からログファイルの追記や一時ファイルへの出力ができずにエラーとなってしまうでしょう。

またサーバ運用時にありがちな例としては、「rootならばエラーなく実行できるだろう」

と安易に考えて、rootユーザでバッチ処理のプログラムを実行してしまうケースです。こうするとログファイルや一時ファイルなどがrootユーザのファイルとして作成されてしまい、次に正規のユーザで実行してもファイルオーナーがrootユーザであるために上書きできず、処理に失敗してしまう、といったケースがよく見られます。

このような理由から、あるプログラムを実行するユーザを制限したい場合があります。そのようなケースで実行ユーザを制限するのが本サンプルです。このサンプル例では、まず❶でプログラム実行を許可するユーザを指定しています。ここではbatch1というユーザに実行許可を与えています。

❷では、現在スクリプトを実行しているユーザが、許可ユーザと一致するかどうかを確認しています。ここで利用しているidコマンドとは、現在のユーザ情報を表示するコマンドです。idコマンドを引数なしで実行すると、ユーザID、グループID、属するグループを表示します。

idコマンドを引数なしで実行した例

```
$ id
uid=500(user1) gid=500(user1) groups=500(user1)
```

❷ではidコマンドに、ユーザIDのみを表示する-uオプションと、IDではなく名前を表示する-nオプションを組み合わせて使っています。このid -nuの結果をコマンド置換$()で取得することで、現在のユーザ名が得られます。この値を、許可ユーザ名を代入しているシェル変数script_userと=演算子で比較しています。

現在のユーザ名と許可ユーザ名が一致すれば、正しい実行ユーザだと判断できますから、❸で外部プログラムを実行しています。また、シェルスクリプトの先頭にこのような処理を入れて、シェルスクリプト自身を実行するユーザを制限する手法もよく使われます。

なお❸で実行している外部プログラムbatch_programとは、何らかのバッチ処理を行う外部プログラムであると仮定しています。読者の環境にあわせて、この外部プログラムは適宜読み替えてください。

現在のユーザ名と許可ユーザ名が一致していない場合は、エラー表示をして終了しています(❹)。このようにしてプログラム実行ユーザを制限することができます。

なお現在のユーザ名を取得するには、idコマンド以外にも、whoamiコマンドを利用する例や環境変数$USERを利用する例があります。どれも得られる結果は同じです。

現在のユーザ名の取得法各種

```
$ whoami
user1
$ echo $USER
user1
```

ユーザ名取得には、これらのコマンドもよく使われますので、覚えておくとよいでしょう。

注意事項

- スクリプトを実行できるユーザを制限するには、ファイルパーミッションを利用した例もよく使われます。例えばスクリプトファイルのパーミッションを754 (-rwxr-xr--) に設定しておくと、ファイルのオーナーとそのグループに所属するユーザしか実行できません。この場合は、バッチを実行するグループを作成して、実行を許可するユーザをそのグループに所属させる、という運用を行います。
- ここではバッチ処理の例を紹介しましたが、実行できるユーザを制限することは、サーバに常駐するデーモンタイプのプログラムでもセキュリティ的に重要です。この場合は、一般ユーザ権限で動作するプログラムを、不用意にrootで起動できないようにしておく手法としてよく使われます。例えばJavaサーブレットのコンテナであるTomcatはrootでも一般ユーザでも起動できますが、root権限で実行することはセキュリティ的に推奨されていません。もし搭載したJavaアプリケーションに脆弱性があった場合、一般ユーザ権限ではなくroot権限が奪われてしまう可能性があり、危険だからです。そのようなプログラムではラッパースクリプト(→P.250)を作り、root権限では実行できないようにしておく手法があります。

関連項目

090 常に指定した環境変数を設定してコマンドを実行するために、ラッパースクリプトを作成する

No. 107 システムのシャットダウンを行う

利用コマンド
who, wc, ps, shutdown

キーワード
ログインユーザ, シャットダウン, プロセス, 確認, 停止

いつ使うか
シャットダウンの手順をスクリプトで記述し、確認項目を自動的にチェックして電源断したいとき

実行例

```
# ./shutdown.sh
[ERROR] whoコマンドの出力が2行以上: 作業中のユーザがいます

# ./shutdown.sh
Broadcast message from user1@linux
        (/dev/pts/1) at 15:32 ...

The system is going down for halt NOW! ―――― システムがシャットダウンされる
```

スクリプト

```
#!/bin/sh

# 自分以外のユーザがログインしていないかを、whoコマンドの出力から
# チェックする
other_user=$(who | wc -l)
if [ "$other_user" -ge 2 ]; then
  echo "[ERROR] whoコマンドの出力が2行以上: 作業中のユーザがいます" >&2
  exit 1
fi

# 事前に停止しておくべきプロセスが、起動したままでないかをチェックする
commname="/usr/libexec/mysqld"
ps ax -o command | grep -q "^$commname"
if [ $? -eq 0 ]; then
  echo "[ERROR] シャットダウンを中止:プロセス $commname が起動中" >&2
  exit 2
fi

# シャットダウンを実行。なおMac/FreeBSDの場合は「注意事項」を参照
shutdown -h now
```

❶ ❷ ❸

解説

このスクリプトは、マシンのシャットダウンを行うものです。シャットダウンの前に、以下の2点をチェックしています。これは実際のサーバ運用時でも、よくあるチェックポイントでしょう。

・他のユーザがログインしていないか
・事前に止めておかないといけないプロセスが起動していないか

マシンのシャットダウンというのは重大な作業のため、手順書を策定することも多いでしょう。しかし手順書だけでは不慣れなユーザが混乱したり、オペレーションミスも起きるかもしれません。

例えば「シャットダウン前に別のユーザがログインしていないか確認すること」と手順書に書いてあっても、実際に何というコマンドを打つかがわからなかったり、正しいコマンドを打っても出力結果を読み間違えるかもしれません。そのため、このサンプル例のようなスクリプトを用意しておいて、「スクリプトshutdown.shでシャットダウンすること」と決めておけば手順書どおりの操作が行えます。

サンプル例では、❶で**whoコマンド**の出力結果を利用して現在のログインユーザをチェックしています。whoコマンドの出力はOSによって若干異なりますが、次のように現在ログイン中のユーザ一覧を表示します。第1カラムがユーザ名、第2カラムが利用している制御端末、第3カラムがログイン日時です。

現在ログインしているユーザを表示

```
$ who
user1    tty1         2014-02-01 11:36
hanako   pts/0        2014-01-29 22:36 (10.211.55.2)
hanako   pts/1        2014-02-01 11:26 (10.211.55.2)
```

ここでは、user1というユーザがシャットダウン操作を行おうとしていると想定します。hanakoというユーザがどうやら外からログインして何か作業しているようですから、何も声をかけずにシャットダウンしてしまうと作業中のデータに大きな影響を与えてしまうかもしれません。

❶ではwhoコマンドの出力が2行以上あれば他のユーザがいると判断する仕様で、別のユーザがログイン中かどうかのチェックを行っています。whoコマンドの出力を、**wcコマンド**の行数のみを出力する**-lオプション**へパイプでつないで、行数をカウントしています。ここではwhoコマンドの行数、すなわち現在のログインユーザ数を、**testコマンド**の演算子**-ge (Greater Than)**で比較しています。このtestコマンドの大小比較の演算子については、P.143で解説していますのでそちらを参照してください。ここでは、もし行数が2以上ならば、別のユーザがログイン中だとみなしてエラーを表示して終了しています。

なお❶のtestコマンドでの大小比較の際、LinuxやFreeBSDのGUI環境やMacを利用している場合には注意が必要です。一般にUNIX環境でX Window SystemなどのGUI環境を利用している場合には、GUI画面にログインしているユーザと、そこからターミナルを使っているユーザの二人いるのが普通のため、3行以上あれば他のユーザがいる、と考えたほうが適当です。すなわち❸は、GUI環境を利用している場合は次のように指定してください。

```
if [ "$other_user" -ge 3 ]; then
```

続いて❷では、シャットダウン前に必要なプロセスのチェックを行っています。ここでは「シャットダウン前に、MySQLを手動で停止すること」という手順を想定しています。このシステムのMySQLを実行するコマンド/usr/libexec/mysqldのプロセスが存在するかを、**psコマンド**と**grepコマンド**で確認しています。プロセスが存在すればgrepコマンドは終了ステータスとして0を返しますから、これをif文で比較してエラーを表示しています。なお、このプロセス確認の手法については、P.313を参照してください。

ここまででチェックが終わりましたので、❸で**shutdownコマンド**を実行しています。ここで**-hオプション**はhalt（停止）を意味するオプションで、システム停止を行います。shutdownコマンドでは、再起動を意味する**-r（reboot）オプション**もよく使われます。

shutdownコマンドは、引数にTIME（時間）を指定して、この時間だけ経った後にシステムを終了させます。TIMEはほとんどの場合、**now**（いますぐ）を指定することが多いでしょう。❸でもnowを指定して、システムを終了させています。

注意事項

- このスクリプトはshutdownコマンドを実行するため、スクリプトはroot権限で実行する必要があります。具体的には、rootユーザで実行するか、MacやUbuntuの場合はsudoコマンドを利用してください。
- shutdownコマンドを実行する際、OSごとに指定方法が多少異なります。FreeBSDでは、shutdownコマンドで電源を切る際には-hオプションではなく-pオプションを利用します。そのため❸の部分を次のように修正してください。

```
shutdown -p now
```

関連項目

054 今年がうるう年かどうかを調べる
111 特定のプロセスが停止していないか監視する

No. 108 ファイル名から、インストールされたRPMパッケージ名を調べる

利用コマンド
rpm, exit

キーワード
RPMパッケージ, インストール

いつ使うか
サーバにあるファイルが、パッケージで入れられたものなのか、手で作られたものなのかわからないケースで、属するRPMパッケージを調べたいとき

実行例

```
$ ./rpm-compkg.sh /etc/ntp.conf
/etc/ntp.conf -> ntp-4.2.4p8-3.el6.centos.x86_64

$ ./rpm-compkg.sh /etc/my.cnf
/etc/my.cnf はパッケージに属していません
```

スクリプト

```
#!/bin/sh

# ファイルを指定するコマンドライン引数をチェック
if [ ! -f "$1" ]; then
  echo "ファイルがありません: $1" >&2
  exit 2
fi

# ファイル名から、属するrpmパッケージ名を取得する
pkgname=$(rpm -qf "$1")

# rpm -qf コマンドの結果でパッケージ名を表示する
if [ $? -eq 0 ]; then
  echo "$1 -> $pkgname"
else
  echo "$1 はパッケージに属していません" >&2
  exit 1
fi
```

❶ ❷ ❸

解説

このスクリプトは、コマンドライン引数にファイルを指定して、そのファイルがパッケージ管理システム上のどのRPMパッケージに属するかを調べて、パッケージ名を表示するものです。なお**RPM**とはレッドハット社が開発したパッケージ管理システムの名称で、

そのインストールなどの操作を行うのがrpmコマンドです。RPMは、Red Hat Linuxや、フリーのCentOSで利用することができます。

Linuxサーバの管理・運用を長いあいだ行っていると、サーバに設置されているあるファイルが、誰かが手で作って置いたものなのか、それともパッケージインストール時に自動的に設置されたものなのか、わからなくなってしまうことがよくあります。このサンプル例は、そのような運用時の調査のシーンで役に立つでしょう。

あるファイルがどのパッケージに属しているかを調べるには、**rpmコマンドの-qfオプション**を利用します。このコマンドは次のように、ファイルを指定することでそのファイルが属するrpmパッケージ名を表示します。

rpmコマンドでファイルが属するパッケージを表示

```
$ rpm -qf /usr/bin/ldd
glibc-common-2.12-1.107.el6_4.5.x86_64
```

また、逆にパッケージ名を指定して、そのパッケージに含まれるファイル一覧を出力することもできます。この際にはrpmコマンドの**-qlオプション**でパッケージ名を指定します。パッケージのバージョン番号は省略可能です。

パッケージに含まれるファイル一覧を表示

```
$ rpm -ql glibc-common
/etc/default
/etc/default/nss
/etc/gai.conf
      :(省略)
```

rpmコマンドはとても高機能なコマンドなので、そのすべての機能を解説していると本書の範囲を大きく超えてしまいます。そのためこのサンプル例の解説では、「**-qfオプションでファイルからパッケージ名を得ることができる**」ということだけの理解にとどめます。

リストの❶では、コマンドライン引数のチェックを行っています。このサンプル例では引数にファイルを指定しますから、そのファイルが存在するかどうかを**testコマンド**でチェックします。ここで、**-f**は対象が通常ファイルかどうかをチェックする演算子です。それを否定演算子!と併用することで、対象がディレクトリであったり、ファイルが存在しない場合にはエラーを表示して終了するようにしています。これらファイルテストの演算子については、P.110で紹介していますのでそちらを参照してください。

❷で、rpmコマンドの-qfオプションを利用してパッケージ名を取得しています。シェルの位置パラメータ$1はコマンドライン引数の1つ目を指しますから、このサンプル例では指定されたファイルパスが入っています。**コマンド置換$()**を用いて、rpmコマンドの出力結果をシェル変数pkgnameに代入しています。

❸で、パッケージ名を表示しています。rpmコマンドの-qfオプションは、指定された

ファイルがどのパッケージにも属していなかった場合はエラーとなり、終了ステータス1を返します。そのためここで、終了ステータス**$?**によってif文で分岐しています。rpmコマンドが成功していた場合はファイル名とそのパッケージ名を表示して、失敗していた場合(すなわち終了ステータスが非ゼロの場合)にはどのパッケージにも属していない旨を表示して終了しています。

なおこのスクリプトでは、引数に指定したファイルがなかった場合は「exit 2」として終了ステータスに2を、ファイルはあるがどのパッケージにも属していなかった場合は「exit 1」として終了ステータスに1を返しています。こうすれば、このスクリプトを別のスクリプトから利用する際、終了ステータスを利用して結果を判別することができます。

注意事項

- このスクリプトはLinux(CentOS)のrpmコマンドを利用しているため、CentOSのみの動作となります。MacとFreeBSDは対象外です。
- CentOSでは、標準rpmパッケージに加えて**EPEL**(Extra Packages for Enterprise Linux)という拡張パッケージが存在します。最新のパッケージを積極的に採用するLinuxディストリビューションであるFedoraのrpmパッケージを、CentOSから利用可能とするのがEPELの仕組みです。CentOS標準パッケージには見つからないソフトウェアも、EPELならば見つかるというケースは多いでしょう。EPELを利用するには、Fedora ProjectのWebページからepel-releaseというrpmパッケージをダウンロードしてインストールします。これにより、リポジトリに追加されたEPELサイトから、**yumコマンド**でEPELのパッケージをインストールできるようになります。詳しい手順をここで解説するのは省略しますが、興味のある方はFedora ProjectのWebサイトからEPELについての説明ページを読んでみてください。

関連項目

042 処理開始前に、実行権限をチェックして正常動作できることを確認してから実行する

No. 109 RPMパッケージ名を記述したリストファイルから、それぞれのパッケージがインストール・更新された日付を調べる

利用コマンド
cat, rpm

キーワード
RPMパッケージ, 更新日時, インストール, アップデート

いつ使うか
複数台のサーバ管理中にパッケージのインストール・アップデートを行う際、サーバごとに作業の漏れがないようチェックしたいとき

実行例

```
$ cat pkg.lst
httpd
zsh
xz
git
$ ./rpm-lastdate.sh pkg.lst
2013年10月14日 11時41分39秒 : httpd
2014年02月01日 18時38分33秒 : zsh
2013年05月18日 02時54分30秒 : xz
2013年12月29日 19時14分14秒 : git
```

スクリプト

```
#!/bin/sh

# 指定されたリストファイルの存在チェック
if [ ! -f "$1" ]; then
  echo "対象のパッケージリストファイルが存在しません: $1" >&2
  exit 1
fi

# 引数で指定されたファイル ($1) から、パッケージリストを取得
pkglist=$(cat "$1")

# インストール済みrpmの更新日付を出力する
rpm -q $pkglist --queryformat '%{INSTALLTIME:date} : %{NAME}¥n'
```

❶ ❷ ❸

解説

このスクリプトは、コマンドライン引数に指定されたテキストファイルからRPMパッケージ名を読み込み、そのRPMパッケージがインストールもしくは更新された日付を一

覧表示するものです。

Linuxサーバを構築・運用していると、何十台もの多くのサーバに対して同一のパッケージをインストールしたり、セキュリティパッチを当てるためにパッケージを更新するメンテナンスを頻繁に行うでしょう。この際、「10台あるサーバのうち1台だけ更新していなかった」「後でやるつもりで1台だけインストールを忘れていた」などという作業漏れが起きるのはありがちな失敗例です。このサンプル例のような確認スクリプトを用意しておけば、メンテナンスの終了確認などの場面で役立つことでしょう。

このスクリプトでは、まず❶で、**位置パラメータ$1**で指定されたリストファイルが存在するかどうかの確認をしています。**-f**は対象が通常ファイルかどうかをチェックする演算子です。それを否定演算子!と併用することで、対象がディレクトリであったり、ファイルが存在しない場合にはエラーを表示して終了するようにしています。これらファイルテストの演算子については、P.110で紹介していますのでそちらを参照してください。

続いて❷で、**コマンドライン引数**で指定されたファイル$1から、シェル変数pkglistにパッケージ名を読み込んでいます。ここで実行例では、pkg.lstファイルはリスト1のような内容であるとしています。

リスト1 パッケージを指定するリストファイル(pkg.lst)の中身

```
httpd
zsh
xz
git
```

つまりここでは、httpd、zsh、xz、gitパッケージのインストール・更新日付を確認したいというケースであると仮定します。読者の環境にあわせて、このパッケージリストの定義は適宜変更してください。

❷では、**catコマンド**でpkg.lstファイルの中身を出力し、**コマンド置換$()**を利用してその出力結果をシェル変数pkglistに代入しています。ここではcatコマンドの結果を用いているので、この時点ではシェル変数pkglistには改行区切りでリスト1そのままの内容が代入されています。

❸で、指定されたRPMパッケージのインストール・更新日付を**rpmコマンド**で表示しています。ここでrpmコマンドに指定しているのは、問い合わせの-qオプションと、表示フォーマットを指定する--queryformatオプションです。

rpmコマンドの**-qオプション**は、対象のパッケージを指定します。この際、複数のパッケージを指定できます。ここではシェル変数pkglistに、❷で組み立てたパッケージリストが入っていますので、これをそのまま指定しています。シェルでは改行をデフォルトの区切り文字として解釈するため、❸のように$pkglistにダブルクォートを付けずに記述すると、改行を区切りとしてシェルが変数展開を行い、結果として次のようなコマンドが実行されることになります。

```
rpm -q httpd zsh xz git --queryformat '%{INSTALLTIME:date} : %{NAME}¥n'
```

この際、❸で誤って"$pkglist"とダブルクォート付きで記述すると、コマンドの途中に改行が入ってしまいエラーとなりますので注意してください。

❸で用いている**--queryformatオプション**は、rpmコマンドで表示する項目を指定するオプションです。続く書式指定の中で使用している**INSTALLTIME**と**NAME**は、それぞれインストール・更新日時、およびパッケージ名に置き換えられます。またデフォルトでは、INSTALLTIMEはUNIX時間(1970年1月1日からの経過秒数)で表示されて読みにくいため、**:date**と指定して日付表示するようにしています。これ以外にもさまざまな項目が表示できますので、興味のある方は項目一覧を表示する**--querytagsオプション**で確認してみてください。

こうしてメンテナンス対象のサーバでサンプル例を実行すれば、各パッケージのインストール・更新日時を表示することができます。例えばあるサーバだけパッケージの更新日が古ければ、パッチのアップデート作業忘れだとわかるでしょう。また、次のようにパッケージが見つからないと表示されたら、インストール作業に漏れがあったと判断できます。このようにしてサーバ構築・運用の作業ミスを減らすことができます。

インストール漏れがあったときの表示例

```
$ ./rpm-lastdate.sh pkg.lst
2013年10月14日 11時41分39秒 : httpd
パッケージ zsh はインストールされていません。
2013年05月18日 02時54分30秒 : xz
2013年12月29日 19時14分14秒 : git
```

注意事項

- このスクリプトはLinux(CentOS)のrpmコマンドを利用しているため、CentOSのみの動作となります。Mac、FreeBSDは対象外です。
- このスクリプトでは、確認するパッケージリストを指定するファイルの中身が空っぽの場合、rpmコマンドが失敗するためエラーとなります。

関連項目

042 処理開始前に、実行権限をチェックして正常動作できることを確認してから実行する
108 ファイル名から、インストールされたRPMパッケージ名を調べる

No. 110 サーバ構築のパッケージリストをシェルスクリプトの形で管理する

利用コマンド
yum

キーワード
RPMパッケージ, インストール, 構築

いつ使うか
サーバの構築作業で、パッケージをインストールする作業を自動化したいとき

実行例

```
# ./yum-install.sh
Loaded plugins: fastestmirror, security
Loading mirror speeds from cached hostfile
      :(省略)       yumコマンドで httpd、zsh、xz、gitがインストールされる
```

スクリプト

```
#!/bin/sh

# インストールするパッケージ名の定義
pkglist="httpd zsh xz git" ―――――――――――――― ❶

# パッケージリストから順に1行ずつ読み込み
for pkg in $pkglist ―――――――――――――――――― ❷
do
  # yumコマンドでパッケージをインストールする
  yum -y install $pkg ―――――――――――――――― ❸
done
```

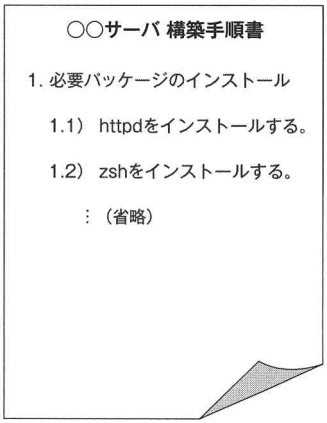
解説

このスクリプトは、シェル変数pkglistで指定されたパッケージを、サーバに自動的にインストールするものです。新規サーバの初期構築に利用して、処理を自動化するケースを想定しています。ここでは、「httpd、zsh、xz、gitをインストールする」という要件であると仮定しています。

このサンプル例でパッケージ管理に利用している**yumコマンド**は、RPMパッケージをインストールするためのコマンドです。依存関係にあるパッケージもあわせて入れてくれるため非常に便利で、現在では単体のrpmコマンドよりも広く使われています。

サーバ構築時、パッケージのインストールをわざわざこのようなスクリプトにしておくのは理由があります。一般的に、このようなインストール作業は、構築手順書として次のようなものを作って、文書どおりに作業するのが昔ながらのやり方でしょう。

◆サーバ構築手順書の例



しかしこのような手順書ベースの作業では、サーバ操作に不慣れな担当者ではどうしてもミスが起こりがちになります。一方、手順書をサンプル例のようにスクリプトの形で作っておけば、別の担当者に交代してもスクリプトを実行すればよいだけですから、誤った操作や作業漏れを防ぐことができます。

つまりこのサンプル例は、サーバ構築の作業をスクリプト化することで、作業の利便性の向上と、作業ミスをなくすという、2つの目的を持たせたものです。

サンプル例では、まず❶でインストールするパッケージを定義しています。ここではhttpd、zsh、xz、gitという4つのパッケージをインストールするように設定しています。このようなスクリプトは、サーバに配布して作業を行うため、リストは外部ファイルにせずにこのようにスクリプトに持たせたほうがよいでしょう。リストを外部ファイルにしてしまうと、リストファイルだけコピーを忘れてしまうなどの作業漏れが発生するかもしれません。

❷のfor文で、シェル変数pkglistから値を1つずつシェル変数pkgに読み込みます。❸で利用しているyumコマンドでは、**install**を指定してパッケージのインストールを行っています。yumコマンドでよく利用する命令を次にあげておくので参考にしてみてください。

◆yumコマンドの主な命令

コマンド	意味
yum install	パッケージのインストール
yum info	パッケージの情報表示
yum list	利用可能なパッケージの一覧を表示
yum search	キーワードでパッケージ検索
yum update	インストール済みのパッケージをアップデート
yum erase	パッケージの削除

なお、yumコマンドでパッケージのインストールを行うと、途中でインストールしてよいかどうかという問い合わせが表示されます。

yumコマンドでインストールすると問い合わせが表示される

```
      :
      :(省略)
      :
Installed size: 2.9 M
Is this ok [y/N]:
```

-yオプションを利用すると、これらの問い合わせにすべて自動的にyを回答します。ここでは自動化スクリプトを作っているわけですから、この-yオプションが必須となります。

このようにしてサーバのパッケージインストールという構築作業を、スクリプトで自動化することができます。読者の環境にあわせて、インストールする具体的なパッケージ名を修正して試してみてください。

注意事項

- このスクリプトはパッケージのインストールを実行するため、スクリプトはroot権限で実行する必要があります。
- このスクリプトはLinux (CentOS) のyumコマンドを利用しているため、CentOSのみの動作となります。Mac、FreeBSDは対象外です。
- サーバ構築の自動化については、最近ではChefやPuppetなどの高機能なソフトウェアが広まりつつあります。これらはまだまだ歴史が浅いソフトウェアのため使用するにはリスクがありますが、パッケージのインストールだけにとどまらず、各種コンフィグファイルの自動設定なども行える、非常に便利なソフトウェアです。サーバ構築の自動化に興味のある方はぜひ調べてみてください。

関連項目

022 ヒアドキュメントで変数展開をせずにそのまま$strのように表示する
108 ファイル名から、インストールされたRPMパッケージ名を調べる

No. 111 特定のプロセスが停止していないか監視する

利用コマンド
ps, grep, wc

キーワード
プロセス, 監視, 停止

いつ使うか
サービス提供の際、存在すべきプロセスが停止していないかを監視したいとき

実行例

```
$ ./process-isalive.sh
[ERROR] プロセス /usr/libexec/mysqld が見つかりません
start alert.sh ....
```

スクリプト

```
#!/bin/sh

# 監視するプロセスのコマンド
commname="/usr/libexec/mysqld" ―――――――――――――――――――――――――――――❶

# 対象コマンドのプロセス数をカウントする
count=$(ps ax -o command | grep "$commname" | grep -v "^grep" | wc -l)
                                                              ❷

# grep コマンドの出力結果がゼロ行の場合にはプロセスが
#存在しないため、監視通知処理を行う
if [ "$count" -eq 0 ]; then
  echo "[ERROR] プロセス $commname が見つかりません" >&2
  /home/user1/bin/alert.sh                                    ❸
fi
```

解説

このスクリプトは、指定したプロセスが存在しているかどうかを監視するものです。ここではMySQLのサーバプロセス、"/usr/libexec/mysqld"が起動しているかどうかを監視して、起動していない場合はプロセスダウンとみなしてalert.shを実行しています。なお、このサンプル例でalert.shは、通知メールを送信するなど、何かしらの警告を行うスクリプトであると仮定しています。読者の環境にあわせて監視するコマンド名を修正し、alert.shでエラーメッセージを表示したり、メールを送信するなど、応用してみてください。

サーバ運用において、プロセスが知らないあいだに停止していることはよくあるトラブル例です。サーバ自体の死活監視をpingコマンドで行っていても、プロセスが落ちていては正常にサービスができません。そのため、プロセス監視の仕組みは、サービス提供のた

めの重要な要素です。ここではシェルスクリプトを用いて、プロセスが存在しているかどうかをチェックするサンプル例を見てみましょう。

サンプル例では、まず❶で監視するプロセスを指定します。このサンプルではコマンド名でプロセスをチェックしますので、Linux (CentOS) でMySQLをrpmでインストールした際の起動コマンド**/usr/libexec/mysqld**を指定しています。

❷では、まず**psコマンド**の**-axオプション**で全プロセスを表示し、さらに**-oオプション**で表示項目を指定しています。ここでは「-o command」と指定しているため、コマンド名の項目のみを表示することになります。

全プロセスのコマンドだけを表示

```
$ ps ax -o command
COMMAND
/sbin/init
[kthreadd]
[migration/0]
     :(省略)
sshd: user1 [priv]
sshd: user1@pts/0
```

これは、プロセス監視では余計な文字列や数値が入っていると誤検知をする恐れがあるため、余計な表示をしないようにするためです。なお、-oオプションはBSDオプション(→後述)ではないため、Linuxで利用する場合もハイフン付きで指定します。

❷のpsコマンドの出力はパイプに渡し、grepコマンドを実行してシェル変数commnameで指定されたコマンド名にマッチするかを調べています。さらにここでは、もう一段パイプでつないで「grep -v "^grep"」としています。これは昔からよく使われる定石のため、読者も前任者などから引き継いだシェルスクリプトに見ることが多いかもしれません。

grepコマンドの**-vオプション**は、そのパターンを含む行を除外するオプションです。なぜこのコマンドを入れているかというと、例えば単純にmysqldプロセスを監視しようと「grep "/usr/libexec/mysqld"」とすると、このgrepコマンド自体がpsコマンドのリストに載ってしまい、誤検知してしまうからです。

psコマンドの結果を絞り込むためのgrepがリストに入ってしまう

```
$ ps ax -o command | grep "/usr/libexec/mysqld"
grep /usr/libexec/mysqld
```

そのため、grepコマンド自体を除去するために、「grep -v "^grep"」を指定しているということです。❷では最後に**wcコマンド**の、行数を数える**-lオプション**を用いてマッチした行数をカウントしてコマンド置換$()でシェル変数countに代入しています。これがすなわち、監視対象のプロセス数になります。

❸で、監視通知をするかしないかの判断をしています。シェル変数countには、grepコマンドでマッチした行、つまりプロセス数が入っています。等しいかどうかを判断する-eq演算子で比較して、これが0ならばプロセスが存在しないことを意味します。そのためプロセスが見つからないというメッセージをechoコマンドで表示し、続けて監視通知を行うalert.shを実行しています。

このようにして、プロセスの監視を行うことができます。このシェルスクリプトをcronに登録し、定期的に実行するのが一般的な使い方となるでしょう。

psコマンドの詳細

psコマンドは、サーバ上で動作しているプロセスを表示するプログラムです。psコマンドには長い歴史があり、オプションの指定も癖があるため、なかなか扱いがわかりにくいコマンドです。このpsコマンドには、大別して2種類のオプション形態があります。

UNIXオプション　：Solarisで利用されていたオプション形式
BSDオプション　：その名のとおり、FreeBSDなどのBSD系列のOSで利用されてきたオプション形式

FreeBSD/Macでは、BSDオプションが利用できます。一方のLinuxでは、この折衷案としてどちらのオプション形式も利用できるようになっています。ただし、Linuxのpsコマンドは「UNIXオプションはハイフン付きで、BSDオプションはハイフンなしで利用する」というルールがあります。また、少々ややこしいのですが、FreeBSDなどのBSD系列のOSでBSDオプションを利用する場合も、ハイフンは省略するのが普通です。

まずUNIXオプションの例を次にあげておきます。

主要なUNIXオプションの例

オプション	説明
-a	端末に関連していないプロセスを除いた、すべてのプロセスを表示
-f	詳細情報を表示（ユーザID、親プロセスIDなど）
-e	現在実行中のすべてのプロセスを表示

続いて、BSDオプションの例を次に示します。ここでは説明のためハイフンを付けていますが、先述したとおり一般的には、実際にコマンドを実行する際はハイフンを付けません。

主要なBSDオプションの例

オプション	説明
-a	自分以外のユーザのプロセス情報も表示
-x	制御端末を持たないプロセスも表示
-l	詳細情報を表示（ユーザID、親プロセスIDなど）

また、両方の形式で使えるオプションもあります。

両形式で使えるオプションの例

オプション	説明
-p	プロセスIDを指定
-u	実行ユーザIDを指定
-o	指定されたキーワードに関するカラムのみを表示

UNIXオプションとBSDオプションのどちらのオプション形式を使うかは好みにもよりますが、現在ではBSDオプションのほうが好まれることが多いようです。そのためこのサンプル例でもBSDオプションを採用し、-aと-xを組み合わせて全プロセスを表示しています。なお先述したように、BSDオプションを利用する場合は基本的にハイフンなしで利用するため、サンプル例の「ax」オプションもハイフンを付けていません。

注意事項

- プロセス監視の際、grepコマンド自体を誤検知しないようにする手法には、サンプル例で利用している「grep -v "^grep"」の他に、次のような手法もよく使われます。

```
ps ax -o command | grep "[/]usr/libexec/mysqld"
```

この手法では、監視したいコマンドの先頭の文字を、文字クラス[]に入れて指定します。ここでは文字クラスの中に/（スラッシュ）しかありませんから、実質的には「grep "/usr/libexec/mysqld"」と書いているのと同じです。しかしこの場合には、psコマンドで取得できるgrepコマンドの引数がカッコ付きとなるため、「/usr/libexec/mysqld」という文字列にはマッチせず、誤検知を防ぐことができます。

- このスクリプトでは、コマンド名で単純にgrepしているため、例えば次のようにコマンド名と同じ文字列を引数にとっているプログラムがある場合には、プロセス有りと誤検知してしまいます。

```
./hoge_check /usr/libexec/mysqld
```

関連項目

114 サーバのping監視を行う

No. 112 特定プロセスの起動本数の閾値チェックを行う

利用コマンド
ps, grep, wc

キーワード
プロセス, 多重起動, 閾値, 本数

いつ使うか
定期的に起動されるコマンドの多重起動を監視し、閾値以上の場合は警告したいとき

実行例

```
$ ./process-numcheck.sh
[ERROR] プロセス /home/user1/bin/calc が多重起動(3)
start alert.sh ....
```

スクリプト

```
#!/bin/sh

# 監視するプロセスのコマンドと、プロセス本数閾値
commname="/home/user1/bin/calc"
threshold=3

# プロセスの本数をカウントする
count=$(ps ax -o command | grep "$commname" | grep -v "^grep" | wc -l)

# プロセス本数が閾値以上ならば、警告処理を行う
if [ "$count" -ge "$threshold" ]; then
  echo "[ERROR] プロセス $commname が多重起動($count)" >&2
  /home/user1/bin/alert.sh
fi
```

❶ ❷ ❸

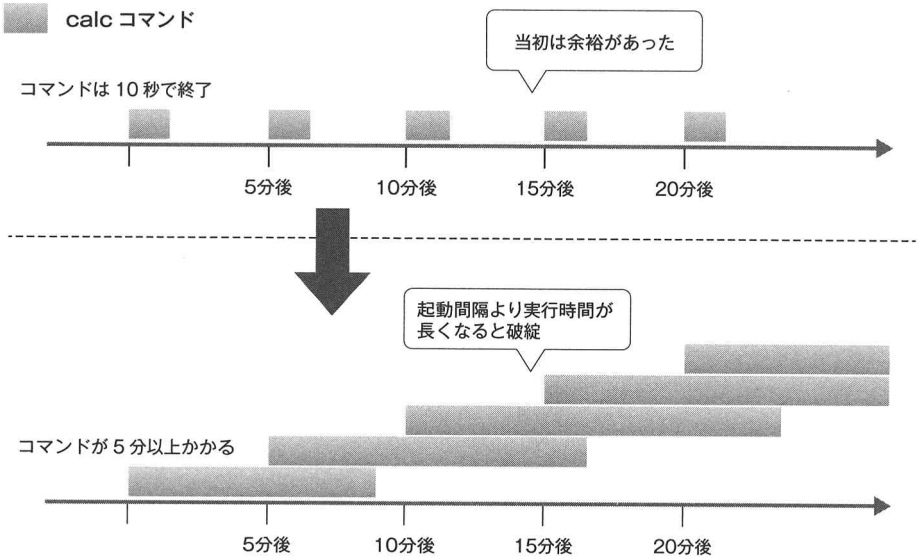
解説

このスクリプトは、指定したプロセスが多重起動していないかどうかを監視するものです。ここではcalcというコマンドの起動本数を監視しています。もしこのcalcプログラムが、3つ以上多重起動されている場合は異常とみなしてalert.shを実行しています。読者の環境にあわせて監視するコマンド名(シェル変数commname)を修正し、alert.shでエラーメッセージを表示したり、メールを送信するなど、応用してみてください。

ここでは前提条件として、calcというプログラムが5分おきに定期的に起動されて、何かしらの計算を行うバッチ処理であるとします。このバッチ処理は、2つまでの多重起動ならば問題ないのですが、3つ以上多重起動すると極端に処理が重くなったり、作業中のデータ破壊をしてしまうなどの問題がある処理と仮定します。

状況として、このcalcコマンドをリリースした当初は、1回の処理に10秒もかからなかったため、定期実行間隔を5分として何も問題ありませんでした。しかし長いあいだシステムの運用をしていると、データベースやログファイルの肥大化によって、バッチ処理時間がどんどん長くなっていく、というのはよくあることです。

当初より負荷が高まって困ることになる



そしてあるとき、calcコマンドの処理時間が5分以上かかるようになってしまうと、5分間隔の起動では処理しきれず、多重起動されたプロセスが積み上がって最悪ではマシンがハングアップしてしまうでしょう。

このようなケースで、プロセスの本数を監視するのがこのサンプル例です。リストでは、まず❶で、監視するコマンドと、プロセス数の閾値を定義しています。読者の環境に応じて、この値を修正してください。短い間隔で定期的に起動するようにcron登録しているバッチ処理などが、対象となるでしょう。

続いて❷で、**psコマンド**を利用してプロセス数を確認します。psコマンドの使い方とプロセスの数を調べるやり方については、P.313で詳しく説明していますのでそちらを参照してください。ここではpsコマンドの結果を、grepコマンドでパイプ処理して、結果の行数を**wcコマンド**の行数を取得する**-lオプション**を利用して取得し、シェル変数countに代入しています。これが現在のプロセス数となります。

❸で、シェル変数countと、最初に定義したプロセス本数の閾値thresholdを比較しています。ここではtestコマンドの「AがB以上かどうか」を判断する演算子**-ge**（以上）で比較しています。この**testコマンド**の大小比較の演算子については、P.143で解説していま

すのでそちらを参照してください。もしプロセス数が3以上ならば、閾値越えとして警告表示を行い、警告処理スクリプトalert.shを実行しています。このようにして、定期起動するコマンドの多重起動をチェックできます。

前ページの図で紹介したバッチ処理の「追い越し・追い付き」は、ときにデータ破壊など重大な障害を招くことがあります。重要な処理を行う場合には、このような監視と共に、ロックファイルなどの利用でそもそも多重起動できないようにするなど、設計段階での計画が必要です。

注意事項

・FreeBSDやMacのwcコマンドで-lオプションを利用すると、先頭にスペースが入るため、次のようにエラーメッセージに余計なスペースが入ってしまいます。

FreeBSDやMacの場合の表示

```
[ERROR] プロセス $commname が多重起動(       3)
```

これが気になる場合は、❷のパイプ処理の最後に、**trコマンド**でスペースを削除する処理を追加するとよいでしょう。

```
count=$(ps ax -o command | grep "$commname" | grep -v "^grep" | ¥
      wc -l | tr -d ' ')
```

・このケースとは逆に、「プロセスが複数あるのが通常だが、ある閾値以下になると異常なので通知したい」というケースもあります。例えば何かしらの常駐型デーモンプログラムで、通常は最低5つのプロセスが立ち上がっているシステムなどでは、プロセスが4つ以下になれば異常としてアラートをあげるべきでしょう。その場合は、このサンプル例の❸で大小比較している部分の「以上(-ge)」を「以下(-le)」と逆にするだけで対応できます。

関連項目

054 今年がうるう年かどうかを調べる
111 特定のプロセスが停止していないか監視する

No. 113 プロセスを監視し、プロセスダウン時に自動的に再起動させる

利用コマンド

service, ps, wc, grep, date, echo

キーワード

プロセス, 監視, 再起動, 自動化

いつ使うか

Webサーバなどの運用時、プロセスがダウンしたことを検知して再起動させ、障害対応を自動化させたいとき

実行例

```
# ./process-restart.sh
[2014/02/03 21:10:51] プロセス /usr/sbin/httpd が見つかりません
[2014/02/03 21:10:51] プロセス /usr/sbin/httpd を起動
Starting httpd: [  OK  ]
```

スクリプト

```
#!/bin/sh

# 監視するプロセスのコマンド
commname="/usr/sbin/httpd" ―――――――――――――――――❶

# 監視プロセスの起動コマンド
start="service httpd start" ―――――――――――――――――❷

# 監視対象コマンドのプロセス数をカウントする
count=$(ps ax -o command | grep "$commname" | grep -v "^grep" | wc -l) ――❸

# grep コマンドの出力結果がゼロ行の場合にはプロセスが
#存在しないため、異常とみなしてプロセスを起動する
if [ "$count" -eq 0 ]; then ―――――――――――――――――❹
  # 日付を入れてログ出力
  date_str=$(date '+%Y/%m/%d %H:%M:%S') ―――――――――❺
  echo "[$date_str] プロセス $commname が見つかりません" >&2
  echo "[$date_str] プロセス $commname を起動" >&2

  # 監視プロセスの起動
  $start ―――――――――――――――――――――――――――❻
fi
```

解説

このスクリプトは、シェル変数commnameで指定されたコマンドのプロセス監視を行い、プロセスが見つからない場合には自動で再起動を行って復旧させるものです。ここでは、Webサーバとしてよく使われる、Apache httpdサーバを対象プロセスと想定します。なお、このスクリプトは、基本的にcronに設定して自動実行されることを仮定しています。

プロセスの監視は、Webサービスを提供するうえでとても重要な要素です。シェルスクリプトでのプロセス監視はP.313で解説したので、ここではその結果を利用して、プロセスを自動再起動させる方法と、その際の注意点について解説します。

サーバ運用において、「自動化」は非常に重要な技術です。読者がはじめてサーバを構築する際は、あまりそのありがたみは感じられないかもしれません。そのサーバはおそらく1台しかなく、夜にプロセスが落ちていても、翌朝に気がついたら起動しなおせばよいレベルのものかもしれません。

しかし運用期間が長くなってくると、えてして利用者の要求は高まり、サーバは何十台にもなります。夜間休日にも、プロセスダウンへの即時対応が求められるようになるかもしれません。また、動いていると思っていたプロセスがいつの間にか落ちていた、という見落としも起きるでしょう。そのような障害への対策として、このサンプル例が役立つかもしれません。

サンプル例では、まず❶で監視するプロセスのコマンドを指定します。ここではApache httpdを対象とします。読者の環境に応じてコマンドのパスは変わるでしょうから、シェル変数commnameは適宜修正してください。

❷では、プロセスの起動を行うコマンドを指定します。ここではLinuxの**serviceコマンド**を使ってhttpdを起動させるよう設定しています。MacやFreeBSDの場合には「注意事項」を参照してください。

❸で、シェル変数commnameで指定したプロセスが存在するかを**psコマンド**で確認します。シェル変数countに現在のプロセス数が代入されているため、これを❹のif文で比較しています。**testコマンド**の**-eq演算子**で0と比較し、0だった場合にはプロセスが存在しないとみなしてif文の中のコマンドが実行されます。

❺では、現在の日付を入れて**echoコマンド**で状態表示をしています。このようなシステム管理のスクリプトでは、後で障害時刻を調べるときのために時間が重要ですから、**dateコマンド**を利用して現在の日時をシェル変数date_strに代入して表示しています。なお、この際のdateコマンドの%Yなどによるフォーマット指定については、P.88で解説していますので詳しくはそちらを参照してください。

❻で監視プロセスの再起動を行っています。事前にシェル変数startで定義しておいたコマンドをそのまま実行しています。こうしてプロセスの停止を検知し、自動的に再起動させることができます。

このようなスクリプトは、「5分に1回」など定期的に動かしたいと思われるでしょう。このようにスクリプトを定期的に動作させるには、cronの仕組みを用います。サンプル例のようにApache httpdを再起動させたい場合は、root権限が必要なため、rootユーザ

のcronとして設定するとよいでしょう。具体的には、/etc/crontabに次のように記述します。

```
*/5 * * * * root /usr/local/bin/process-restart.sh >> /var/log/myapp/start.
log 2>&1
```

この例では分指定を「*/5」と記載しているため、5分おきにプロセスチェックが行われます。

なお、ここでは対象のプロセスとしてApache httpdをサンプルにあげました。ApacheなどのWebサーバは、リクエストに対してコンテンツを返すだけのソフトウェアであり、このようにある程度気軽に再起動をしてもかまわないケースが多いでしょう。

一方、例えばMySQLなどのデータベースサーバに、このような自動再起動スクリプトを安易に適用するのは危険です。データベースサーバが予期せぬ停止をした場合には、ディスク容量があふれたりI/Oエラーが多発しているなど、何かしらの理由があるのが普通です。その原因を調べずにこのような自動スクリプトで強制的に再起動を繰り返してしまうと、最悪の場合、データベースの破壊という致命的な障害を招いてしまうかもしれません。

そのため、MySQLサーバなどのデータベースサーバでは、プロセス監視はするべきですが、このサンプル例のようなプロセス異常時に再起動させるスクリプトまで適用させるのは注意が必要です。プロセスダウンなどの異常時にはその状態を保持しておき、手動復旧を前提とするべきシステムもあるため、自動再起動をさせるかどうかは事前によく検討しましょう。

一般的には、Webサーバやプロキシサーバなどは自動再起動をしてもあまり問題にはなりません。一方、データベースサーバの自動再起動を行う際は、きちんとした事前検証と設計が必要でしょう。

注意事項

- このサンプル例では、Linux（CentOS）でApache httpdをrpmインストールした際に提供される起動スクリプトを再起動に用いました。FreeBSDやMacでApache httpdの起動スクリプトを用意していない場合、apachectlコマンドを直接実行するとよいでしょう。例えばMacならば/usr/sbinにapachectlコマンドがあるため、❷を次のように修正します。

```
start="/usr/sbin/apacectl start"
```

関連項目

033 ファイルをバックアップする際にファイル名に日時を入れる
111 特定のプロセスが停止していないか監視する

No. 114 サーバのping監視を行う

利用コマンド
ping, sleep, date

キーワード
サーバ監視, ネットワーク, 終了ステータス

いつ使うか
ネットワークの状態に異常はないか、あるいはサーバが落ちていないかを、pingコマンドを利用して監視したいとき

実行例

```
$ ./ping_alert.sh 192.168.2.1
[2014/02/03 11:19:43] Ping OK: 192.168.2.1
```

スクリプト

```
#!/bin/sh

# ping実行結果のステータス。0で成功とみなすため、1で初期化する
result=1 ―――――――――――――――――――――――――――――――――――❶

# 対象サーバがコマンドライン引数で指定されていなければ
# エラーとして終了
if [ -z "$1" ]; then
  echo "対象ホストを指定してください" >&2
  exit 1
fi ―――――――――――――――――――――――――――――――――――――――❷

# pingコマンドを3回実行するループ。成功したらresultを0とする
i=0 ―――――――――――――――――――――――――――――――――――――――❸
while [ $i -lt 3 ]
do
  # pingコマンドを実行。終了ステータスのみが必要なため、
  #表示は/dev/nullへリダイレクトして捨てる
  ping -c 1 "$1" > /dev/null ―――――――――――――――――――――――――❹

  # pingコマンドの終了ステータスを判断。成功ならresult=0としてループから抜ける。
  # 失敗ならば3秒のウェイトを入れて再実行
  if [ $? -eq 0 ]; then ―――――――――――――――――――――――――――――❺
    result=0
    break
  else
    sleep 3
    i=$(expr $i + 1)
```

```
  fi
done

# 現在日付を「2013/02/01 13:15:44」の形で組み立て
date_str=$(date '+%Y/%m/%d %H:%M:%S')  ❻
# pingの実行結果を$resultから判断して表示する
if [ $result -eq 0 ]; then
  echo "[$date_str] Ping OK: $1"
else
  echo "[$date_str] Ping NG: $1"
fi  ❼
```

解説

このスクリプトは、**pingコマンド**でサーバが正常に稼働しているかの監視を行うものです。コマンドライン引数に対象サーバのIPアドレスまたはホスト名を指定して実行することで、対象サーバへのping結果をOK/NG表示します。これにより、対象サーバのping応答を監視して、サーバが正常に稼働しているかどうかを判断できるでしょう。

pingコマンドでは対象のサーバを指定して、**ICMP**というプロトコルでパケットを送信します。ICMPにはいくつかのメッセージタイプがありますが、pingコマンドではEcho Requestパケットを送信して、その応答のEcho Replyが返ってくるかどうかで通信状態を確認します。pingコマンドでよく使われるオプションを、次に示します。

pingコマンドの主なオプション

オプション	説明
-c <count>	count個のICMPパケットを送信した後に停止する
-i <秒数>	パケットを送るたびに、指定秒数だけ待つ。デフォルトは1秒
-q	quietモード。開始時と終了時のみメッセージを出力し、途中経過は表示しない

なお、シェルスクリプトでpingコマンドにより対象ホストを監視する場合には、多少の工夫が必要です。例えば、ホストは正常でもネットワークの状態などでたまたま応答がないことがあります。1回エラーになっただけで監視異常のアラートをあげてしまうと、実は正常に動いていたという誤検知が多くなります。そのため、実用上は「**数回投げて1発も返ってこなければ異常あり**」と判断するのが妥当です。サンプル例でも1、2回失敗しても無視して、3回連続失敗ならば異常とみなす、という運用を仮定しています。

サンプルの❶で、まずpingコマンドの実行結果ステータスを格納するシェル変数resultを定義しています。このサンプル例では成功した場合に0とすることにして、ここではまず1で初期化します。

❷で、**testコマンド**の**-z演算子**を利用してコマンドライン引数の対象ホストが指定されているかどうかをチェックしています。**$1**はコマンドライン引数の1つ目の値が入る、シェルの特殊変数です。-zは空文字列ならば真となるため、この❶のif文が真となる場合は、

引数が指定されていません。そのためif文の中では「対象ホストを指定してください」とエラー出力して、exit 1としてエラー終了させています。

❸が、pingコマンドを実行するwhileループです。ここでシェル変数iはループカウンタで、0に初期化しています。

❹で、pingコマンドの**-cオプション**を利用して、1回だけpingを実行しています。この終了ステータスがシェルの特殊変数$?に入っているため、if文で0かどうかを判断します(❺)。終了ステータスが0ならば、pingコマンドは成功ですからシェル変数resultを0(成功)として、break文でwhileループから抜けます。一方、pingコマンドの終了ステータスが非ゼロ(すなわちpingコマンドが失敗、Echo Replyが返ってこないなど)の場合は、**sleepコマンド**で3秒待ってから再びpingを繰り返します。

❻で、**dateコマンド**を用いて現在の時刻を「2013/02/01 13:15:44」のようなフォーマットで組み立てています。このような監視スクリプトでは、問題発生時に後で確認するときのために、OK/NGを判断した日時を一緒に入れておくことが重要です。そのためこの時刻を後で出力時に利用します。なおこのdateコマンドでのフォーマット指定については、P.88で解説していますので詳しくはそちらを参照してください。

最後に❼で、pingコマンドの結果を出力しています。正常か異常かは、シェル変数resultを用いて0ならば正常、1ならば異常と判断しています。

こうして、対象ホストの稼働監視をpingコマンドで行うことができます。定期的に実行するようにcronなどに登録し、失敗の場合はアラートメールを送るスクリプトを実行するなど、読者の環境にあわせて修正してみてください。

注意事項

- このスクリプトを利用する前には、まず対象のサーバにpingコマンドを手で実行してみて、正常な応答が返ってくることを確認してください。最近のOSではセキュリティを考慮して、デフォルトではpingコマンドに応答を返さない設定になっている場合があります。
- このスクリプトのテストをする際は、あるサーバがpingに応答したりしなかったりという状況が作りたくなるでしょう。例えばLinuxでは、次のコマンドでカーネルパラメータを一時的に変更すると、ping応答をしないように設定できるため、簡単にテストできます。

```
#サーバがping応答をしないようにする
echo "1" > /proc/sys/net/ipv4/icmp_echo_ignore_all

#サーバがping応答をするようにする
echo "0" > /proc/sys/net/ipv4/icmp_echo_ignore_all
```

関連項目

033 ファイルをバックアップする際にファイル名に日時を入れる
057 pingで特定ホストへの応答平均時間を取得する

No. 115 Webアクセス監視を行う

利用コマンド
curl, date, echo

キーワード
Web監視, サービス監視, HTTPステータスコード

いつ使うか
運用しているWebサービスについて、アクセステストを定期的に実行し、異常時には警告通知をあげたいとき

実行例

```
$ ./web-curlcheck.sh
[2014/02/06 17:50:32] HTTPステータス異常：HTTP status[503]
ALERT...
```

スクリプト

```
#!/bin/sh

# 監視対象URLを指定する
url="http://www.example.org/webapps/check" ―――――――――――――――――❶

# 現在日付を「2013/02/01 13:15:44」の形で組み立て
date_str=$(date '+%Y/%m/%d %H:%M:%S') ―――――――――――――――――――――❷

# 監視URLにcurlコマンドでアクセスし、終了ステータスを変数curlresultに代入
httpstatus=$(curl -s "$url" -o /dev/null -w "%{http_code}") ―――――――――❸
curlresult=$?

# curlコマンドが失敗していればHTTP接続自体が異常とみなす
if [ "$curlresult" -ne 0 ]; then ―――――――――――――――――――――――❹
  echo "[$date_str] HTTP接続異常：curl exit status[$curlresult]"
  /home/user1/bin/alert.sh
# 400番台、500番台のHTTPステータスコードならばエラーとみなして警告
elif [ "$httpstatus" -ge 400 ]; then ―――――――――――――――――――――❺
  echo "[$date_str] HTTPステータス異常：HTTP status[$httpstatus]"
  /home/user1/bin/alert.sh
fi
```

解説

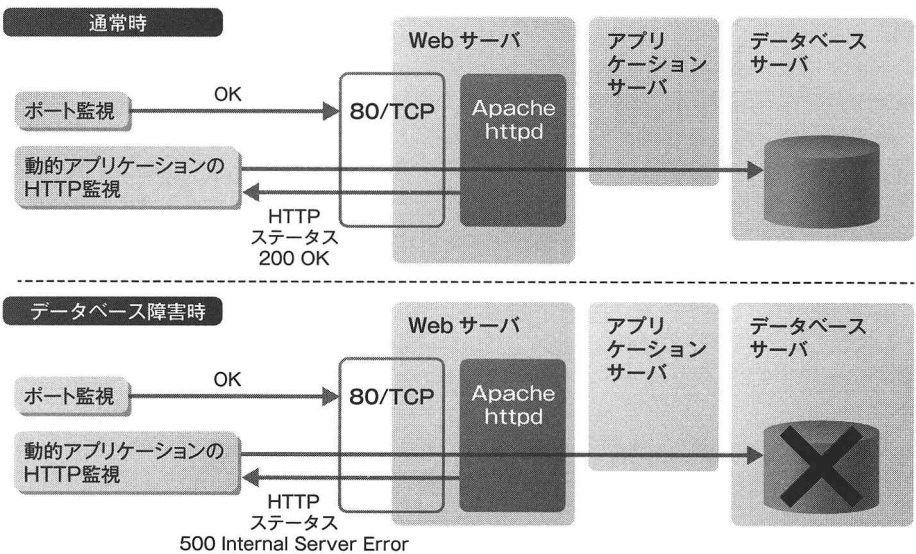
このスクリプトは、Webサーバへのアクセス監視を行って、異常時には警告を発生させるものです。ここでは異常時にはalert.shというスクリプトを実行し、これで通知を行

うものと仮定しています。読者の環境に応じて、このalert.shの中身をメール送信するよう設定するなど、適宜修正してください。

このサンプル例では、**curlコマンド**で**HTTPステータスコード**を確認しています。そのため単にWebサーバが稼働しているかどうかだけでなく、そこで稼働しているアプリケーション状態まで監視できることがポイントです。なお、このようなWebサービス監視について実際にどのように運用すべきかについては、APPENDIXのP.410に詳しく説明していますので、そちらを参照してください。

Webサービスを提供するにあたっては、wgetコマンドなどでHTTP接続できるかどうかだけのテストでは不十分です。例えば、次のようにバックエンドのデータベースに障害が発生したケースを考えます。

バックエンドサーバでの障害も含めてテストしたい



ここでもしポート監視しか行っていないと、データベース障害時もWebサーバには正しくTCPポートの80番で接続できるため、障害を見逃してしまいます。HTTPステータスコードまで監視していれば、WebサーバはHTTPステータスコード500 (Internal Server Error) などを返すでしょうから、アプリケーションが正しく動作していないことを検知できます。

そのため、Webアプリケーションの監視ではHTTPステータスコードまでを見る監視が必要です。サンプル例ではこのチェックを行っています。

まず❶で、監視対象のURLをシェル変数urlに定義しています。ここで対象のURLは、前図で見るようにバックエンドサーバまで問い合わせるアプリケーションのパスを指定したほうがよいでしょう。

❷では、**dateコマンド**を用いて現在の時刻を「2013/02/01 13:15:44」のようなフォーマットで組み立て(→P.51)、これを警告表示の時刻へと利用しています。

❸で、curlコマンドによりHTTPステータスコードを取得してシェル変数httpstatusに代入しています。curlコマンドにはさまざまなオプションがありますが、ここでは次にあげた3つのオプションを利用しています。

スクリプトで使用したcurlコマンドのオプション

コマンド	説明
-s	silentモード(静かなモード)。途中経過などを表示しない
-o	取得したファイルの出力先を指定
-w	コマンド完了後に出力する表示フォーマット指定

このうち❸では、**-wオプション**で%{http_code}と指定して、HTTPステータスコードを出力しています。これ以外にも-wオプションでは、トータルでかかった時間やダウンロードサイズ、Content-Typeなどさまざまな値を指定できます。詳しく知りたい方は、man curlとしてcurlコマンドのマニュアルを読んでみてください。

また❸では、あわせてcurlコマンド自体の終了ステータス$?もシェル変数curlresultに代入しています。これは、何らかのネットワーク障害や、URLの誤りでcurlコマンド自体が失敗した場合のチェックも行いたいためです。このcurlコマンドの終了ステータス確認は❹のif文で行っており、curlコマンド自体が失敗している場合には「HTTP接続異常」としてエラーメッセージを表示し、警告をあげるスクリプトalert.shを実行しています。

❺では、HTTPステータスコードにより正常か異常かを判断しています。一般に、HTTPステータスコードは400番台および500番台が異常系のコードです。よく見られる異常系のHTTPステータスコードを次に示します。

チェックすべき異常系HTTPステータスコード

コード	一般的な表示	説明
400	Bad Request	リクエストが不正。存在しないメソッドなど
403	Forbidden	アクセス拒否。サーバの設定で接続拒否しているなど
404	Not Found	ファイルが見つからないなど
500	Internal Server Error	サーバの内部エラー。CGIでのエラーなど
502	Bad Gateway	Proxyサーバなどにおいて、上位サーバから不正なレスポンスを受け取った
503	Service Unavailable	サーバがビジー状態などで処理を受け付けることができない

例えば503 (Service Unavailable)は、サーバが高負荷状態となったときによく見られます。また500 (Internal Server Error)は、プログラムに何らかのバグがあり、正常にレスポンスを返せない場合にも発生します。

これら400番台および500番台のHTTPステータスコードが返ってきた場合は異常とみなしたいため、❺では**testコマンド**の**-ge演算子**でHTTPステータスコードが400以上かど

うかを判断しています。400以上ならば、異常であるとみなしてメッセージを表示し、警告をあげるスクリプトalert.shを実行しています。このように、HTTPステータスコードでの異常判断は、コード値が400、もしくは500以上かどうかで行うのが比較的ポピュラーです。

このようにして、Webサービスの監視を行うことができます。cronに登録するなどして定期的に実行されるようにして、読者の運用するサービスに利用してみてください。

注意事項

- FreeBSDには、curlコマンドがデフォルトではインストールされていません。FreeBSD標準のfetchコマンドではHTTPステータスコードを取得できないため、次の手順でportsからcurlコマンドをインストールしてください。

portsでcurlコマンドをインストール

```
# cd /usr/ports/ftp/curl
# make install
```

- Webサーバを運用していると、Webクローラや、あるいは何らかの悪意のある攻撃者がさまざまなアクセスをしてくるのが普通です。そのためアクセスログファイルには、多くの404 (Not Found) が見つかることでしょう。これをあまり気にする必要はありませんが、攻撃の徴候がつかめたり、ページのリンク切れが見つかることもあるため、アクセスログファイルも定期的にチェックしたほうがよいでしょう。

No. 116 ディスクの容量監視を行う

利用コマンド
df, awk, read, echo, rm

キーワード
ディスク, 使用量, 使用率, 容量

いつ使うか
ディスクの使用率を定期的に監視して、閾値以上の使用率となったときに警告を出したいとき

実行例

```
$ ./df-diskcheck.sh
[2013/02/01 13:15:44] Disk Capacity Alert: /usr/local (92% used)
ALERT...
```

スクリプト

```
#!/bin/sh

# 監視するディスク使用量の閾値パーセンテージ
used_limit=90 ―――――――――――――――――――――――――――――――❶
# dfコマンドの出力結果一時ファイル名
tmpfile="df.tmp.$$" ―――――――――――――――――――――――――――――❷

# dfコマンドでディスク使用量を表示。1行目はヘッダなので除外する
df -P | awk 'NR >= 2 {print $5,$6}' > "$tmpfile" ―――――――――――――❸

# dfコマンドの出力の一時ファイルから、使用率を確認する
while read percent mountpoint ―――――――――――――――――――――――❹
do
  # "31%" を "31" に、末尾の%記号を削除する
  percent_val=${percent%%%} ―――――――――――――――――――――――――❺

  # ディスク使用量が規定値以上ならばアラート
  if [ "$percent_val" -ge "$used_limit" ]; then ―――――――――――――――❻
    # 現在日付を「2013/02/01 13:15:44」の形で組み立て
    date_str=$(date '+%Y/%m/%d %H:%M:%S') ―――――――――――――――――❼

    echo "[$date_str] Disk Capacity Alert: $mountpoint ($percent used)"
    /home/user1/bin/alert.sh
  fi
done < "$tmpfile"

# 一時ファイル削除
rm -f "$tmpfile" ―――――――――――――――――――――――――――――――❽
```

解説

このスクリプトは、サーバがマウントしているディスクについてそれぞれ使用率を監視して、指定した値よりも使用率が大きくなっているディスクがある場合に警告を発するものです。ここでは**dfコマンド**でディスクの空き容量を調べ、シェル変数used_limitで指定された閾値パーセンテージの値より使用率が大きい場合には、alert.shというスクリプトを実行して警告を発生させます。

なお、このサンプル例でalert.shは、通知メールを送信するなど、何かしらの警告を行うスクリプトであると仮定しています。読者の環境に応じて、このalert.shでメールを送信するなど、適宜修正してみてください。

サーバ運用に当たって、ディスク使用量は見落としやすい監視ポイントです。サービス開始当初はディスクの空き容量が大きいため、その使用量は誰も気にしませんし、監視を怠ることも多いでしょう。しかし何年ものあいだサービスを行っていると、ログファイルや一時ファイル、アプリケーションの出力したデータファイルなどが積み重なってディスクを圧迫します。そしてある日突然サーバがダウンし、調べてみるとディスク使用量が100%となっていた…というのはありがちな失敗例です。

そこで、サーバ運用の当初から、このサンプル例のようなディスク監視スクリプトをcronに仕掛けておくと、将来何かの保険として役立つでしょう。

サンプルの❶では、まずこのスクリプトが監視するディスク使用率の閾値を、シェル変数used_limitに定義しています。ここで設定したパーセンテージの値よりもディスクの使用率が高ければ、警告を発します。例えばused_limitに90を設定していて、対象ディスクの容量が50GBならば、90%すなわち45GB以上ディスクを使用していると警告を発生させます。

❷では、dfコマンドの出力を保存する一時ファイルのファイル名を定義しています。ここではシェルの特殊変数**$$**を利用して、プロセスIDをファイル名に用います。この$$の利用例については、P.129を参照してください。

❸で、dfコマンドの出力を**awkコマンド**で処理して使用率を抽出します。dfコマンドはディスクの使用量などを表示するコマンドで、オプションなしの出力例は次のようになります。

dfコマンドの実行例

```
$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/vda3            100893076   2195932  93571976   3% /
tmpfs                   510208         0    510208   0% /dev/shm
/dev/vda1               247919     51510    183609  22% /boot
nfsdisk001.example.com:/export/home
                      34600237   8869759  22270454  28% /mnt/nfs
```

出力は左列より、「ファイルシステム・ディスクの全容量・ディスクの使用量・ディスクの空き容量・ディスクの使用率(%)・マウントポイント」となります。またdfコマンドには、次のようなオプションがあります。

dfコマンドの主なオプション

オプション	意味
-h	人間が読みやすい形式(human-readable)でサイズを表示する。例えば2ギガバイトならば「2G」と表示する
-k	1024バイト(1キロバイト)単位でサイズを表示する
-i	iノード情報を表示する
-P	POSIX出力形式を用いる。ファイルシステム名が長くても改行を入れずに1行で表示する
-l	ローカルファイルシステムのみを対象とし、NFSなどは表示しない

サンプルでは**-Pオプション**を利用しています。先ほどの実行例の最終行を見ればわかるように、ファイルシステム名が長い場合、dfコマンドは自動的に改行を入れてしまいます。シェルスクリプトで扱う際にはこれでは不都合ですから、-Pオプションを利用して改行しないようにしているのです。

❸では、まずdfコマンドの1行目はヘッダのため不要ですので、awkコマンドで「NR >= 2」とフィルタ指定して2行目以降を表示しています。なお、**NR**とはawkの組み込み変数で、現在の処理行を指します。そしてディスクの使用率とマウントポイントを取得するため、第5カラム($5)と第6カラム($6)をawkコマンドでprintしています。結果を後で使いますから、一時ファイル$tmpfileにリダイレクトしています。ここまでの結果として、一時ファイル$tmpfileの中身は、次のようになっています。

リスト1 一時ファイルの内容

```
3% /
0% /dev/shm
22% /boot
28% /mnt/nfs
```

❹で、一時ファイル$tmpfileの中身をwhile文に入力リダイレクトして読み込んでいます。**readコマンド**を用いて、シェル変数percentおよびmountpointに、それぞれ利用率とマウントポイントを代入します。

❺では、利用率の文字列からパーセント記号(%)を消しています。前ページの図で見たように利用率はパーセント表示されますが、これは数値比較する際に邪魔なため、削除して数値のみにします。なお❺ではパーセント記号(%)の削除に、シェルのパラメータ展開を利用しています。

シェルの**パラメータ展開**とは、${parameter%word}と書くことで、「変数parameterの値から、wordに後方一致でマッチする部分を削除した値」を得る機能です。❺では%を削除したいのですが、パラメータ展開では%自体がメタキャラクタであるため、エスケープ

して${percent%¥%}と記述しています。これで「シェル変数percentの、一番後ろの%を削除した値」という意味になります。なお、このパラメータ展開を利用した文字列処理については、P.62の注意事項で説明していますので、そちらもあわせて参照してください。

これでディスクの使用率が数値として得られたため、❻では**testコマンド**の「以上」を意味する**-ge演算子**で、閾値と比較しています。使用率が閾値以上ならば警告を表示して、アラートを発生させるスクリプトalert.shを実行しています。なお❼では、**dateコマンド**を用いて現在の時刻を「2013/02/01 13:15:44」のようなフォーマットで組み立てています。このdateコマンドでのフォーマット指定については、P.88で解説していますので詳しくはそちらを参照してください。この結果は、警告表示の時刻へと利用しています。

最後に❽で、dfコマンドの一時ファイルを削除しています。これを行わないと、一時ファイルがいつまでも消えずに残ってしまいディスクを圧迫してしまうので注意してください。

このようにして、ディスク使用率を監視することができます。cronに登録して、定期的に実行するよう設定しておくとよいでしょう。

注意事項

- FreeBSDやMacでは、特殊なデバイスファイルシステムdevfsやmapのCapacity（使用率）が常に100%となっています。これは監視に不都合ですので、除外したほうがよいでしょう。具体的には、❸のdfコマンドを次のように変更してください。

```
df -P | grep -v '^devfs¥|map ' | awk ' NR >= 2 {print $5,$6}' >
"$tmpfile"
```

関連項目

023 絶対パスで起動されても相対パスで起動されても、同じ動作をできるようにする
033 ファイルをバックアップする際にファイル名に日時を入れる
049 二重起動が可能な一時ファイル作成する

No. 117 メモリ・スワップ監視を行う

利用コマンド
vmstat, awk, date, echo

キーワード
メモリ, 監視, スワップ

いつ使うか
メモリの空き状態を定期的に監視して、スワップが発生しているときに警告を出したいとき

実行例

```
$ ./swapcheck.sh
[2014/02/06 22:40:18] Swap Alert: 352 (si+so)
ALERT...
```

スクリプト

```
#!/bin/sh

# 監視するスワップ発生回数。これ以上ならば警告する
swapcount_limit=10 ―――――――――――――――――――――――――――――――――❶

# vmstatコマンドの出力からスワップイン・スワップアウト値を取得する
swapcount=$(vmstat 1 6 | awk 'NR >= 4 {sum += $7 + $8} END{print sum}') ―❷

# スワップ回数が閾値を越えていれば警告
if [ "$swapcount" -ge "$swapcount_limit" ]; then ――――――――――――――❸
  # 現在日付を「2013/02/01 13:15:44」の形で組み立て
  date_str=$(date '+%Y/%m/%d %H:%M:%S') ――――――――――――――――――――❹

  # スワップ発生の警告出力
  echo "[$date_str] Swap Alert: $swapcount (si+so)" ――――――――――――――❺
  /home/user1/bin/alert.sh
fi
```

解説

このスクリプトは、現在のサーバのメモリ状態を確認し、メモリ不足が起きていないかどうかを監視するものです。ここでメモリ不足とは、サーバで発生しているスワップイン・スワップアウトの回数で判断する仕様とします。頻繁にスワップが発生しているようであれば、メモリ不足と判断して警告メッセージを表示します。

スワップ回数は**vmstatコマンド**で取得し、シェル変数swapcount_limitで指定された値よりも直近5秒間のスワップイン・スワップアウトの回数が大きい場合には、alert.sh

というスクリプトを実行しています。なお、このサンプル例でalert.shは、通知メールを送信するなど、何かしらの警告を行うスクリプトであると仮定しています。読者の環境に応じて、このalert.shの中身は適宜修正してみてください。

❶では、まずスワップ回数の監視閾値を設定しています。このスクリプトでは、1秒おきに5回計測したスワップイン・スワップアウトの回数をすべて足した値が閾値以上ならば、警告を行うとします。ここでのスワップ回数の閾値10というのは、ときどき警告が出るくらいなら問題ないでしょうが、常時警告がでるようならば問題だろうと考えています。実際の閾値は読者の環境にあわせて修正してみてください。一般的には、Apache httpdなどのWebサーバで静的コンテンツを返すだけの場合には、ディスク上のコンテンツをリクエストに対して送り出すだけで、あまりメモリが必要な処理はありません。そのためスワップは発生せずに0が普通です。一方、Apache httpdで多くのCGIを動かしているWebサーバや、Javaアプリケーションなどを動作させているアプリケーションサーバは多くのメモリを使うため、スワップの発生には注意を払う必要があります。

❷で、スワップの回数を取得するためにvmstatコマンドを実行しています。vmstatコマンドは、サーバの現在のリソース状態を表示するコマンドです。OSによって表示形式はかなり異なるため、ここではLinuxの例を紹介します。FreeBSDやMacについては後述します。

vmstatコマンドの実行例（Linuxの場合）

```
$ vmstat 1 3
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0      0 202940 132208 543284    0    0     0     0    1    0  0  0 100  0  0
 0  0      0 202932 132208 543284    0    0     0     0   41   14  0  0 100  0  0
 0  0      0 202932 132208 543284    0    0     0     0   32   12  0  0 100  0  0
```

上記ではvmstatコマンドの引数に「1 3」を指定しています。最初の引数がインターバルで、何秒ごとに計測するかを指定します。2つ目の引数が何回計測するかで、ここでは1秒ごとに3回計測しています。vmstatコマンドではさまざまなデータがとれますが、ここでは「---swap---」列のみに注目します。**si**がスワップイン、**so**がスワップアウトの回数です。そのためこの第7カラムと第8カラムの数値を足した値を指標とします。

なお、vmstatコマンドの1行目、最初の出力だけは、現在の状態ではなくサーバ起動時からの平均が出力されます。ですからこの1行目は、現在の状態とかけ離れた値がでることがしばしばです。そのため監視などの用途では、vmstatコマンドの1行目は読み飛ばすのがセオリーです。

❷では、**awkコマンド**で「NR >= 4」とフィルタして、ヘッダの2行ぶんとコマンド1行目の出力をスキップし、4行目からデータ取得するようにしています。そしてスワップ回数のsiとsoは第7カラムと第8カラムにありますから、この$7と$8の値をsumというawkの変数に足しています。最後にENDブロックでsumを出力することで、直近5秒間のスワップ回数を表示できます。このスワップ回数は、コマンド置換$()を用いてシェル変数

swapcountに代入しています。

❸で、計測したスワップ回数が、事前に定義していた閾値を上回っているかどうかをif文で判断して分岐しています。**testコマンド**の「以上」を意味する**-ge演算子**で、シェル変数swapcount（スワップ回数）とシェル変数swapcount_limit（閾値）を比べています。

スワップ回数が閾値より大きければ、まず❹で、**dateコマンド**を用いて現在の時刻を「2013/02/01 13:15:44」のようなフォーマットで組み立てています。このような監視スクリプトでは、問題発生時に後で確認するときのために、日時を一緒に入れておくことが重要です。なお、このdateコマンドでのフォーマット指定については、P.88で解説していますので詳しくはそちらを参照してください。

最後に❺で、警告表示を行ってスワップ発生回数を出力しています。このようなスクリプトを、サーバのメモリ状態を監視するためにcronに登録して定期的に実行してみるとよいでしょう。

FreeBSDやMacの場合

FreeBSDのvmstatコマンドは、Linuxと表示形式が異なります。

FreeBSDのvmstat実行例

```
% vmstat 1 3
 procs      memory      page                    disks     faults         cpu
 r b w     avm    fre   flt  re  pi  po    fr  sr ad0 cd0   in   sy   cs us sy id
 1 0 0    490M   420M   686   1   1   0   718 819   0   0   20  253  236  1  3 97
 0 0 0    490M   420M     1   0   1   0     0   0   1   0    3  127  137  0  0 100
 0 0 0    490M   420M     0   0   0   0     0   0   0   0    1  117  119  0  0 100
```

FreeBSDではsi/soではなく、**pi**（ページイン）/**po**（ページアウト）でスワップ発生回数を取得できます。これは第8カラムと第9カラムにありますから、❷を次のように修正してください。

```
swapcount=$(vmstat 1 6 | awk 'NR >= 4 {sum += $8 + $9} END{print sum}')
```

MacにはvmstatコマンドはなくVM、代わりに**vm_statコマンド**が存在します。これは使い方も出力もLinuxとはかなり異なります。

Macのvm_stat実行例

```
$ vm_stat -c 3 1
Mach Virtual Memory Statistics: (page size of 4096 bytes, cache hits 0%)
  free active   spec inactive   wire   faults     copy    0fill reactive  pageins  pageout
226719 274773  93077   265596 122511  108973K  2488901 69631334   594279  1598690   209423
226960 272018  92998   265543 125173     1341        7      869        0        0        0
225657 276034  93783   265620 121881     6073      720     2336        0       28        0
```

Macのvm_statコマンドでは、計測回数は-cオプションで指定して、インターバルはコマンドライン引数の1つ目に指定します。なお-cオプションは最近のバージョンで付加されたもので、昔のMacOSでは利用できないようです。

Macのvm_statコマンドでは、FreeBSDと同様に、ページイン(**pageins**)とページアウト(**pageout**)の回数を取得すればスワップの発生回数がわかります。ただしMacのvm_statコマンドは出力形式もバージョンによって異なるようで、10.9 (Mavericks)ではカラム構成が大きく変わっています。そのため読者の環境でvm_statコマンドを実行し、pageinsとpageoutのカラム位置を調べて、❷を修正してみてください。

注意事項

- このようにメモリ監視をシェルスクリプトで行うのは、小規模環境で補助的に用いるケースが多いでしょう。もっと大規模で本格的な監視システムの構築には、APPENDIXのP.410で紹介しているZabbixやNagiosの利用を検討してみてください。
- Javaアプリケーションを動かしている場合のメモリ監視としては、スワップだけでなくガベージコレクション(特にFull GC)の頻度もパフォーマンスに影響します。これはJVM(Java仮想マシン)の起動オプション「-verbose:gc」でログ出力できます。本書では詳しい解説は省略しますが、興味のある方はJavaの専門書を読んでみてください。

関連項目

030 あるディレクトリ内の、n日前からm日前までに更新されたファイル一覧を取得する
033 ファイルをバックアップする際にファイル名に日時を入れる

No. 118 CPU使用率の監視を行う

利用コマンド
mpstat, tail, awk, echo, date, iostat

キーワード
CPU, 負荷, アイドル値, 利用率, 監視

いつ使うか
CPU負荷を定期的に監視して、アイドル値が低下しているときに警告を出したいとき

実行例

```
$ ./cpu-idlecheck.sh
[2014/02/07 08:48:18] CPU %idle Alert: 4.53 (%)
ALERT...
```

スクリプト

```
#!/bin/sh

# 監視するCPU %idleの閾値。この値以下ならば警告
idle_limit=10.0 ―――――――――――――――――――――――――――――――――――――❶

# CPUの%idleをmpstatコマンドで取得。最終行の平均値を取り出す
cpu_idle=$(mpstat 1 5 | tail -n 1 | awk '{print $NF}') ―――――――――――――――❷

# 現在の%idleと閾値をbcコマンドで比較
is_alert=$(echo "$cpu_idle < $idle_limit" | bc) ―――――――――――――――――――❸

# 警告アラートをあげるか判断
if [ "$is_alert" -eq 1 ]; then ―――――――――――――――――――――――――――――❹
  # 現在日付を「2013/02/01 13:15:44」の形で組み立て
  date_str=$(date '+%Y/%m/%d %H:%M:%S') ―――――――――――――――――――――――――❺

  # CPU %idle 低下の警告出力
  echo "[$date_str] CPU %idle Alert: $cpu_idle (%)" ―――――――――――――――――❻
  /home/user1/bin/alert.sh
fi
```

解説

このスクリプトは、直近5秒間のCPUの使用率を調べて、CPU負荷を監視するものです。ここでCPU負荷とは、CPUのアイドル値である%idle (CPUがアイドル状態＝未使用状態となっていた時間の割合)を計測する仕様とします。

サンプル例では**mpstatコマンド**でCPU %idleを取得し、シェル変数idle_limitで指定さ

れた値よりも、直近5秒間の%idle平均値が小さい場合には、CPUリソースが足りないと判断してalert.shというスクリプトを実行しています。なお、このサンプル例でalert.shは、通知メールを送信するなど、何かしらの警告を行うスクリプトであると仮定しています。読者の環境に応じて、このalert.shの中身は適宜修正してみてください。

アプリケーションサーバなどでは、プログラム内に予期せぬ無限ループがあったり、重い処理を長いあいだ実行するなどした際、サーバの負荷が高まりCPU処理が追い付かない場合があります。このようなCPUリソース不足のケースではサーバ全体の処理速度が落ち、ユーザへの応答時間も長くなってしまいます。CPUがシステムのボトルネックになっていないかどうかをモニタをするために、このようなサンプル例が役に立つでしょう。

CPUの利用状態を取得するには、Linuxではmpstatコマンドが便利です。mpstatコマンドの出力は次のようになります。

mpstatの実行例

```
$ mpstat 1 3
Linux 2.6.32-358.23.2.el6.x86_64 (centos)  02/07/14  _x86_64_  (2 CPU)

09:00:29     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest   %idle
09:00:30     all   36.50    0.00   13.50    0.00    0.00    0.00    0.00    0.00   50.00
09:00:31     all   37.19    0.00   12.56    0.00    0.00    0.00    0.00    0.00   50.25
09:00:32     all   13.43    0.00    4.48    0.00    0.00    0.00    0.00    0.00   82.09
Average:     all   29.00    0.00   10.17    0.00    0.00    0.00    0.00    0.00   60.83
```

上記ではmpstatコマンドの引数に「1 3」を指定しています。最初の引数がインターバルで、何秒ごとに計測するかを指定します。ここでは1秒ごとです。2つ目の引数が何回計測するかで、ここでは3回計測しています。

多くの項目がありますが、CPUの負荷を簡易的に見たいならば最終行の平均値(Average)の**%idle**だけ見れば十分でしょう。この値が極端に低ければ、CPU負荷が高い状態であると言えます。サンプル例でも、この値を利用します。

ではサンプル例を見ていきましょう。リストの❶では、まずCPU %idleの監視閾値を設定しています。この値よりも現在のCPU %idleの値が小さければ、CPU負荷が高いと判断し、警告を発する仕様です。ここでは10.0%とします。

❷で、mpstatコマンドの出力から、直近5秒のCPU %idleの平均値を取得しています。上記実行例で見たように、最終行の一番右側のカラム(図では60.83)に取得したいCPU %idleの平均値が出力されますから、ここを取り出します。

❷ではまずmpstatコマンドを実行し、パイプでつないだ**tailコマンド**の行数指定オプション-nで1を指定して、最終行のみを表示します。そして**awkコマンド**で「{print $NF}」とアクション指定することで最後のカラムのみを表示しています。**$NF**とはawkの変数で、最後のカラムを意味します。

❸では、取得したCPU %idleが、事前に定義した閾値以下かどうかを判定しています。シェルスクリプトでの数値比較は、整数ならばexprコマンドで簡単に行えます。しかし

exprコマンドは小数を扱えないため、ここでは**bcコマンド**を用いています。bcコマンドは、次のように大小比較の式を**echoコマンド**で標準入力に与えることにより、比較結果を出力できます。この際、真は1、偽は0となることに注意してください。

bcコマンドでの大小比較

```
$ echo "1.1 > 2.5" | bc
0 ―――――――――――――――――――「1.1は2.5より大きいか?」は偽なので0
$ echo "1.1 < 2.5" | bc
1 ―――――――――――――――――――「1.1は2.5より小さいか?」は真なので1
```

これで❸によりシェル変数is_alertに警告をあげるかどうかのフラグがセットできたため、❹のif文で分岐しています。is_alertが真(値が1)ならばCPU %idleは閾値以下ですから、if文の中の警告処理が実行されます。

❺では、**dateコマンド**を用いて現在の時刻を「2013/02/01 13:15:44」のようなフォーマットで組み立てています。このような監視スクリプトでは、問題発生時に後で確認するときのために、日時を一緒に入れておくことが重要です。なお、このdateコマンドでのフォーマット指定については、P.88で解説していますので詳しくはそちらを参照してください。

最後に❻で、警告表示を行ってCPU %idle値を出力しています。このようなスクリプトを、サーバのCPU負荷状態を監視するためにcronに登録して定期的に実行してみるとよいでしょう。

FreeBSD／Macの場合

FreeBSDおよびMacにはmpstatコマンドがないため、代わりに**iostatコマンド**を利用してください。iostatコマンドの出力例は次のようになります。

FreeBSDのiostatコマンド

```
$ iostat 1 3
       tty           vtbd0             vtbd1             cpu
 tin  tout  KB/t tps  MB/s   KB/t tps  MB/s  us ni sy in id
   0     6 27.66  12  0.32  59.39  24  1.37   9  0  4  0 87
   0   183  0.00   0  0.00   0.00   0  0.00  11  0  3  0 86
   0    61  0.00   0  0.00  38.04 135  5.01   5  0  3  0 92
```

Macのiostatコマンド

```
$ iostat 1 3
          disk0           disk1       cpu     load average
    KB/t tps  MB/s     KB/t tps  MB/s  us sy id   1m   5m   15m
   59.11   5  0.30   199.60   0  0.00   7  2 91  0.14 0.17 0.21
```

```
    0.00    0  0.00      0.00    0  0.00    3  2 96  0.13 0.17 0.21
    0.00    0  0.00      0.00    0  0.00    3  3 94  0.13 0.17 0.21
```

上記のようにMacとFreeBSDで多少差はありますが、CPU %idleは、cpu項目のidという値です。FreeBSDでは最終カラムにあります。一方、Macでは後ろから数えて4番目のカラムにあります。iostatコマンドの出力カラムはマウントしているディスクの数によって変わるため、CPU利用率をとりたいときはこのように後ろから数えるのが適当です。

なおiostatコマンドは、vmstatコマンド(→P.335)と同様に、最初の1行目は現在値ではなくシステム起動時からの平均値が表示されます。そのため監視の際には1行目を無視するのが普通です。

iostatコマンドはmpstatコマンドと違い、平均値を出力してくれないので自分で計算する必要があります。awkコマンドでidカラムの数値を足して、処理行数で割って平均値を求めるとよいでしょう。具体的には❷の部分を次のように修正します。ここでは「iostat 1 6」として、計測値を6回出力しますが、1行目を無視して5つの計測値を用います。そのためawkコマンドで平均値を求める際も5.0で割り算しています。

```
#FreeBSDの場合
cpu_idle=$(iostat 1 6 | awk 'NR >= 4 {sum += $NF} END{print sum/5.0}')

#Macの場合
cpu_idle=$(iostat 1 6 | awk 'NR >= 4 {sum += $(NF-3)} END{print sum/5.0}')
```

iostatコマンドでのCPU %idelの平均値計算は、6回出力して1行目は無視しています。またMacの場合は「後ろから4カラム目」を抜き出すために、最終カラムを表すawkの変数NFから3を引いています。

注意事項

- Linuxでは、インストールオプションによってはmpstatコマンドがデフォルトでは存在しない場合があります。コマンドが見つからないときは、mpstatコマンドが含まれるsysstatパッケージをインストールしてください。
- mpstatコマンドで表示されるCPUの%idle値などは、複数のCPUがある場合にはその平均になります。例えばCPUが2つあるマシンで、マルチプロセッサ対応していないプロセスが片方だけのCPUリソースを圧迫している場合には、片方のCPUはidleが0%(フル稼働)、もう片方のCPUはidleが100%(何もしていない)ということになるでしょう。しかしmpstatコマンドでは平均値しか出ないため、この場合にはidleは50%と表示されます。そのようなケースでCPUごとのCPU使用率を表示したい場合は、mpstatコマンドでCPUごとの使用率を表示する**-P ALLオプション**を利用してみてください。

マルチプロセッサの場合のコマンド使用例

```
$ mpstat -P ALL 1 3
      :(省略)
Average:      CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest   %idle
Average:      all   36.38    0.00   13.62    0.00    0.00    0.00    0.17    0.00   49.83
Average:        0   72.76    0.00   27.24    0.00    0.00    0.00    0.00    0.00    0.00
Average:        1    0.00    0.00    0.00    0.00    0.00    0.00    0.66    0.00   99.34
```

・このようにCPUのidle監視をシェルスクリプトで行うのは、小規模環境で補助的に用いるケースが多いでしょう。もっと大規模で本格的な監視システムの構築には、APPENDIXのP.410で紹介しているZabbixやNagiosの利用を検討してみてください。

関連項目

033 ファイルをバックアップする際にファイル名に日時を入れる
117 メモリ・スワップ監視を行う

No. 119 Webページの変更監視を行う

利用コマンド
curl, cmp, echo, data

キーワード
URL, ファイル変更, 監視

いつ使うか
不定期に変更されるWebサイトの内容を監視し、ファイルに変更があったら通知してほしいとき

実行例

```
$ ./url-diffcheck.sh
[2014/02/08 15:13:30] 前回ダウンロード時からファイル変更がありました
対象URL: http://www.example.org/update.html
```

スクリプト

```
#!/bin/sh

# 監視対象のURL
url="http://www.example.org/update.html" ―――❶

# ダウンロードファイルのファイル名定義
newfile="new.dat" ┐
oldfile="old.dat" ┘――――――――――――――――❷

# ファイルをダウンロード
curl -so "$newfile" "$url" ―――――――――――――❸

# 前回ダウンロードしたファイルと、❸でダウンロードしたファイルを比較
cmp -s "$newfile" "$oldfile" ―――――――――――❹

# cmpコマンドの終了ステータスが非ゼロならば差があった
if [ $? -ne 0 ]; then ―――――――――――――――❺
  # 現在日付を「2013/02/01 13:15:44」の形で組み立て
  date_str=$(date '+%Y/%m/%d %H:%M:%S') ―――――❻

  # ファイル変更の通知
  echo "[$date_str] 前回ダウンロード時からファイル変更がありました" ┐
  echo "対象URL: $url"                                              ├―❼
  /home/user1/bin/alert.sh                                          ┘
fi

# ❸でダウンロードしたファイルをリネームして保存
mv -f "$newfile" "$oldfile" ―――――――――――――❽
```

解説

このスクリプトは、指定されたURLからファイルをダウンロードして、そのファイルが前回のスクリプト実行時にダウンロードしたファイルと異なる場合に通知するものです。通知の際は、alert.shというスクリプトを実行しています。なお、このalert.shは、通知メールを送信するなど、何かしらの警告を行うスクリプトであると仮定しています。読者の環境に応じて、このalert.shの中身は適宜修正してみてください。

あるURLで何かの数値データなどが公開されており、それが不定期に更新されているとします。そのような際は、定期的にファイルをダウンロードして、前回ぶんから更新されていないかどうかを自動的にチェックできると便利です。このサンプル例では、そのような特定のURLの変更監視をするケースを想定しています。1日1回など、定期的に実行してみるとよいでしょう。

リストの❶では、対象のURLを指定してシェル変数urlに代入しています。このスクリプトでは変更比較には、バイナリファイルも扱えるcmpコマンドを使っています。そのためここで指定するURLには、HTMLファイルなどのテキストベースのファイルだけでなく、JPG画像などバイナリファイルを指定してもかまいません。

❷で、ダウンロードしたファイルを保存する際の、ローカルでのファイル名を指定します。スクリプト実行時に保存するファイル名をシェル変数newfile、前回実行時に保存していたファイル名をシェル変数oldfileで定義しています。

❸では**curlコマンド**を使って対象URLからファイルをダウンロードします。curlコマンドには、何も表示しない**-s (slient) オプション**と、出力ファイル名を指定する**-oオプション**を利用しています。ここまでで、シェル変数urlで指定されたURLから、シェル変数newfileで指定されたファイル名でファイルを保存したことになります。

❹で、いまダウンロードしたファイルと前回ダウンロードしたファイルをcmpコマンドで比較しています。**cmpコマンド**は2つのファイルの中身を比較するコマンドです。2つのテキストファイルの差分をとるdiffコマンドと違って、ファイルを1バイトずつ比較するため、バイナリファイルも対象とすることができます。

cmpコマンドは、終了ステータスにより次のような意味を持ちます。

cmpコマンドの終了ステータス

終了ステータス	意味
0	2つのファイルに違いはない
1	2つのファイルに違いがある
2	指定されたファイルが見つからないなど、エラーが発生した

なお❹ではcmpコマンドの終了ステータスのみを利用するため、何も出力をしない**-s (silent) オプション**を指定しています。

❺で、cmpコマンドの**終了ステータス$?**を利用してif文で分岐しています。ここでは終了ステータスが非ゼロであった場合は差があったとみなし、if文の中のコマンドを実行す

るわけです。

❻では、**dateコマンド**を用いて現在の時刻を「2013/02/01 13:15:44」のようなフォーマットで組み立てています。このdateコマンドでのフォーマット指定については、P.88で解説していますので詳しくはそちらを参照してください。この現在時刻を利用して、❼でファイルに変更があった旨の通知を表示しています。このalert.shは、メールを送信するなど、読者の環境にあわせて適宜修正してください。

最後に❽で、今回ダウンロードしたファイルを保存するためにmvコマンドでリネームしています。前回ぶんのファイルに上書きすることになるため、上書き確認をしない-fオプションを付けています。

このようにして、URLで指定したファイルの変更監視をすることができます。cronに登録して、1日1回など定期的に実行してみるとよいでしょう。

注意事項

- もし対象のファイルサイズが大きくディスク容量が不安な場合は、ファイルのハッシュ値のみを保存しておくとよいでしょう。具体的には、P.288で紹介したMD5値だけを保存しておき、前回ぶんと比較する手法です。ハッシュ値を比較すればファイルに変更があったかどうかわかりますから、ファイル自体を保存しておく必要がなく、ディスク容量を節約できます。
- はじめてこのスクリプトを実行する場合は、前回ダウンロードしたファイルがないためにcmpコマンドが失敗するため、必ず「変更がありました」と表示される仕様です。
- FreeBSDではcurlは標準でインストールされていません。そのため次のように、代わりにfetchコマンドを利用するとよいでしょう。

```
fetch -qo "$newfile" "$url"
```

関連項目

033 ファイルをバックアップする際にファイル名に日時を入れる
103 Webサーバからファイルをダウンロードして、ファイルのMD5ハッシュ値を計算する

No. 120 MySQLデータベースのバックアップ

利用コマンド

mysqldump, date, gzip, find, xargs

キーワード

MySQL, データベース, バックアップ, ダンプファイル

いつ使うか

MySQLデータベースのバックアップを、定期的に自動取得したいとき

実行例

```
$ ./mysql-dbbackup.sh
        /home/user1/backupにバックアップファイルが作成される
```

スクリプト

```
#!/bin/sh

# データベース接続設定
DBHOST="192.168.11.5"
DBUSER="backup"
DBPASS="PASSWORD"
DBNAME="hamilton"                                              ❶

# データベースバックアップ設定
BACKUP_DIR="/home/user1/backup"
BACKUP_ROTATE=3
MYSQLDUMP="/usr/bin/mysqldump"                                 ❷

# バックアップ出力先ディレクトリのチェック
if [ ! -d "$BACKUP_DIR" ]; then                                ❸
  echo "バックアップ出力先ディレクトリが存在しません: $BACKUP_DIR" >&2
  exit 1
fi

# 今日の日付をYYYYMMDDで取得
today=$(date '+%Y%m%d')                                        ❹

# mysqldumpコマンドでデータベースのバックアップを取得
$MYSQLDUMP -h "${DBHOST}" -u "${DBUSER}" -p"${DBPASS}" "${DBNAME}" >
"${BACKUP_DIR}/${DBNAME}-${today}.dump"                        ❺

# mysqldumpコマンドの終了ステータス$?で、成功・失敗を確認
if [ $? -eq 0 ]; then                                          ❻
```

```
  gzip "${BACKUP_DIR}/${DBNAME}-${today}.dump"

  # 古いバックアップファイルを削除する
  find "$BACKUP_DIR" -name "${DBNAME}-*.dump.gz" -mtime +${BACKUP_ROTATE} |
xargs rm -f ―――――❼
else
  echo "バックアップ作成失敗: ${BACKUP_DIR}/${DBNAME}-${today}.dump"
  exit 2
fi
```

解説

このスクリプトは、**MySQL**サーバから指定したデータベースのバックアップを作成してファイルとして保存するものです。取得したバックアップファイルはgzip圧縮して保存し、さらに古いバックアップファイルは自動的に削除します。

ここではMySQLデータベースがすでに稼働しており、❶で定義した設定でデータベースに正常に接続できているものと仮定します。MySQLのインストールや設定については本書の範囲を超えますので、専門の書籍などを参照してください。

❷では、データベースのバックアップ設定を定義しています。ここで定義しているシェル変数は次のような意味を持ちます。

スクリプトで使用しているバックアップ設定に関するシェル変数

変数	説明
BACKUP_DIR	バックアップファイルの保存先ディレクトリ
BACKUP_ROTATE	過去何世代ぶんのバックアップファイルを保存するか
MYSQLDUMP	mysqldumpコマンドのフルパス

❸ではシェル変数BACKUP_DIRで指定された、バックアップファイルの出力ディレクトリが存在するかどうかを確認しています。-dは対象がディレクトリかどうかをチェックする演算子です。それを否定演算子!と併用することで、出力先が存在しなかったりディレクトリでない場合には、エラーを表示して終了するようにしています。これらファイルテストの演算子については、P.110で紹介していますのでそちらを参照してください。

❹では、ファイル名に使うために現在の日付をYYYYMMDD形式で**dateコマンド**を用いて取得します。例えばいまが2014年2月3日ならば、20140203が得られます。このdateコマンドでのフォーマット指定については、P.88で解説していますので詳しくはそちらを参照してください。

❺で実際にデータベースのダンプを取得します。ここで**mysqldumpコマンド**に指定しているオプションは、次のとおりです。

スクリプトで使用したmysqldumpコマンドのオプション

オプション	説明
-h	MySQLサーバのホスト名
-u	MySQLサーバに接続するためのユーザ名
-p	MySQLサーバに接続するためのパスワード

❺のmysqldumpコマンドの出力は、リダイレクトして日付付きのファイル「${DBNAME}-${today}.dump」として出力しています。このようにファイル名に日付を入れておくと、後で利用する際にいつのバックアップかがわかるため便利です。

❻で、mysqldumpコマンドが成功したか失敗したかで処理を分岐しています。終了ステータス$?が0ならば正常にコマンドは終了していますから、まず取得したダンプファイルをgzipコマンドで圧縮しています。mysqldumpコマンドで取得できるダンプファイルの中身は単なるテキストファイルのため、データベースの規模によっては大変にサイズが大きくなってしまいます。そのためこのように圧縮して保存しておいたほうがよいでしょう。

❼では**findコマンド**の**-mtimeオプション**を利用し、シェル変数BACKUP_ROTATEで指定された日数より前に作られたバックアップファイルを**xargsコマンド**で削除しています。ここで利用しているfindコマンドの-mtimeオプションとxargsコマンドについては、P.79とP.82で詳しく解説しているためそちらを参照してください。

このようにして、MySQLデータベースのバックアップを自動的に取得することができます。cronに設定して、1日1回定期的に実行しておくとよいでしょう。

注意事項

- mysqldumpコマンドを実行するには、接続時のMySQLアカウントに適切な権限が設定されている必要があります。一般的には、SELECT/SHOW VIEW/LOCK TABLES権限が必要ですが、mysqldumpコマンドで指定するオプションによってはさらに追加の権限が必要になります。詳しくはMySQLの専門書を参照してください。
- このスクリプトにはMySQLサーバへの接続パスワードが書かれているため、ファイルのパーミッションに注意して、他のユーザからは不用意にファイルの中身を参照できないようにしてください。また、データベースのバックアップファイルは、データベースの全データが入っている非常に重要なファイルです。そのためこの保存先ディレクトリも、他のユーザからは読めないように適切なパーミッションを設定するべきです。一般に、データベースサーバのセキュリティ確保に比べ、バックアップサーバのセキュリティ対策はどうしても手薄になりがちです。バックアップサーバには、データベースサーバと同等か、それ以上のセキュリティ対策が必要だということはしっかりと認識しておくようにしてください。

関連項目

030 あるディレクトリ内の、n日前からm日前までに更新されたファイル一覧を取得する
033 ファイルをバックアップする際にファイル名に日時を入れる
042 処理開始前に、実行権限をチェックして正常動作できることを確認してから実行する

No. 121 MySQLのレプリケーション監視

利用コマンド
mysql, awk, grep, data

キーワード
MySQL, レプリケーション, 監視

いつ使うか
MySQLのレプリケーション構成において、レプリケーション状態が壊れていないかを定期的にチェックしたいとき

実行例

```
$ ./mysql-replcheck.sh
[2014/02/08 18:55:00] STATUS NG
Slave_IO_Running: No
Slave_SQL_Running: Yes
Last_IO_Error: Got fatal error 1236 from master when reading data from
binary log: 'Could not find first log file name in binary log index
file'
Last_SQL_Error:
```

スクリプト

```
#!/bin/sh

# データベース接続設定。スレーブサーバに接続する
DBHOST="192.168.11.5"
DBUSER="operator"
DBPASS="PASSWORD"

# mysqlコマンドのフルパス指定と一時ファイルの定義
MYSQL="/usr/bin/mysql"
resulttmp="tmp.$$"

# SHOW SLAVE STATUSをMySQLサーバに問い合わせて、一時ファイルに出力
$MYSQL -h "${DBHOST}" -u "${DBUSER}" -p"${DBPASS}" -e "SHOW SLAVE STATUS
¥G" > $resulttmp

# レプリケーション状態に関するパラメータを抽出
Slave_IO_Running=$(awk '/Slave_IO_Running:/ {print $2}' "$resulttmp")
Slave_SQL_Running=$(awk '/Slave_SQL_Running:/ {print $2}' "$resulttmp")
Last_IO_Error=$(grep 'Last_IO_Error:' "$resulttmp" | sed 's/^  *//g')
Last_SQL_Error=$(grep 'Last_SQL_Error:' "$resulttmp" | sed 's/^  *//g')

# 現在日付を「2013/02/01 13:15:44」の形で組み立てて用意
```

❶ ❷ ❸ ❹

```
date_str=$(date '+%Y/%m/%d %H:%M:%S') ―――❺

# Slave_IO_Running と Slave_SQL_Running が共に YES でなければエラー
if [ "$Slave_IO_Running" = "YES" -a "$Slave_SQL_Running" = "YES" ]; then
  echo "[$date_str] STATUS OK"
else
  echo "[$date_str] STATUS NG"
  echo "Slave_IO_Running: $Slave_IO_Running"
  echo "Slave_SQL_Running: $Slave_SQL_Running"
  echo "$Last_IO_Error"
  echo "$Last_SQL_Error"

  # 警告メール送信などのアラートをあげる
  /home/user1/bin/alert.sh
fi

# 一時ファイルの削除
rm -f "$resulttmp" ―――❼
```

❻

解説

このスクリプトは、**MySQL**サーバの**レプリケーション**(複製)状態を監視するものです。MySQLのスレーブサーバに接続し、レプリケーション異常となっていれば警告メッセージを表示します。レプリケーションの意味については、P.352を参照してください。

このスクリプトでは**mysqlコマンド**で"SHOW SLAVE STATUS"というMySQLのコマンドを発行し、レプリケーション異常と判断すれば警告メッセージを表示してalert.shというスクリプトを実行しています。なお、このサンプル例でalert.shは、通知メールを送信するなど、何かしらの警告を行うスクリプトであると仮定しています。読者の環境に応じて、このalert.shの中身は適宜修正してみてください。

ここではMySQLのレプリケーションがすでに稼働しており、❶で定義した設定でデータベースに正常に接続できているものと仮定します。MySQLのレプリケーション構築については本書の範囲を超えますので、専門の書籍などを参照してください。

❷ではmysqlコマンドのフルパスと、一時ファイルのファイル名を定義しています。一時ファイル名に用いている**$$**という変数は、プロセスIDを指します。この$$の利用例については、P.128を参照してください。

❸でMySQLサーバに対して、mysqlコマンドで"SHOW SLAVE STATUS"命令を発行しています。ここでmysqlコマンドに指定しているオプションを、次に示します。

スクリプトで使用しているmysqlコマンドのオプション

オプション	説明
-h	MySQLサーバのホスト名
-u	MySQLサーバに接続するためのユーザ名
-p	MySQLサーバに接続するためのパスワード
-e	指定されたSQLを実行

-eオプションを使うと、シェルのコマンドラインから直接MySQLサーバに命令を実行できます。この-eオプションを利用して❸で実行している**SHOW SLAVE STATUS**は、レプリケーション構成のスレーブサーバにてレプリケーション状態を表示するコマンドです。出力例は次のようになります。

SHOW SLAVE STATUSの出力

```
             Slave_IO_State: Waiting for master to send event
                Master_Host: 192.168.11.5
:(省略)
    Relay_Master_Log_File: mysql-bin.000151
         Slave_IO_Running: Yes
        Slave_SQL_Running: Yes
:(省略)
           Last_IO_Errno: 0
           Last_IO_Error:
          Last_SQL_Errno: 0
          Last_SQL_Error:
:(省略)
```

Yesなら正常

上記では多くの部分を省略しています。レプリケーション状態を確認したいときにチェックすべきなのは、このうち**"Slave_IO_Running"**と**"Slave_SQL_Running"**です。この2つの値が共に"Yes"となっていなければ、レプリケーションに何かしらの問題が起きています。❸では、このSHOW SLAVE STATUSの出力結果を一時ファイルにリダイレクトして保存しています。

❹で、一時ファイルに出力されているSHOW SLAVE STATUSの結果から、必要な値を取得してシェル変数に代入しています。Slave_IO_RunningとSlave_SQL_Runningについては、**awkコマンド**のフィルタを利用して2カラム目($2)を取得しています。またそれらのエラーメッセージであるLast_IO_ErrorとLast_SQL_Errorについて、**grepコマンド**で一時ファイルから抽出しています。この際には、行頭の余分なスペース記号を削除するために**sedコマンド**も利用しています。

❺では、**dateコマンド**を用いて現在の時刻を「2013/02/01 13:15:44」のようなフォーマットで組み立てています。このdateコマンドでのフォーマット指定については、P.88で解説していますので詳しくはそちらを参照してください。この現在時刻を利用して、❻でメッセージ出力時に日時を入れ込んでいます。

❻が、レプリケーション状態を確認して警告をあげるかどうかを判断するif文です。レプリケーション状態が正常であるかどうかは、Slave_IO_RunningとSlave_SQL_Runningが共にYesであるかで判断します。ここではtestコマンドの=演算子でそれぞれの文字列を比較し、それを**-a演算子**でつないでANDで比較しています。もしこの結果が真ならばレプリケーション状態には異常なしと判断し、OKを出力するだけとしています。

もし❻の結果が偽ならば、レプリケーション状態を異常とみなします。例えばマスターサーバのバイナリログが見つからない場合は、次のようにSlave_IO_RunningはNoとなり、Last_IO_Errorにエラーメッセージが表示されます。

レプリケーション異常時のSHOW SLAVE STATUS

```
Slave_IO_Running: No
Slave_SQL_Running: Yes
Last_IO_Error: Got fatal error 1236 from master when reading data from
binary log: 'Could not find first log file name in binary log index file'
Last_SQL_Error:
```

❻のelse節の中では、レプリケーション異常であることをNGと表示し、エラーメッセージを同時に表示しています。またalert.shを実行し、警告メッセージを発しています。読者の環境にあわせて、このalert.shでメールを送信するなど適宜修正してみてください。最後に❼で、一時ファイルを削除しています。

このようにして、レプリケーション状態の監視を行うことができます。現在のMySQLシステムにおいて、レプリケーションを構築する機会は多く、読者の運用管理するシステムで扱うこともあることでしょう。レプリケーションによってシステムの可用性は高められますが、データ不整合を産む原因となることもあるため、定期的にこのようにレプリケーション状態を監視することは運用上とても重要です。

データベースとレプリケーション

レプリケーションとはデータベースサーバで利用されるシステム構成で、次図のようにマスターサーバからスレーブサーバへデータを複製することです。このようにしてシステムの可用性を高めたり、データベースサーバの負荷軽減を図ることができます。

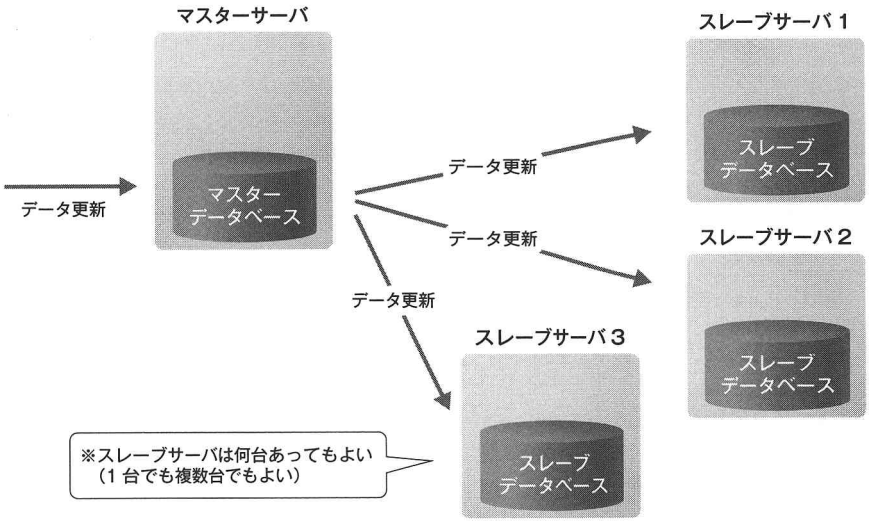
例えばこのデータベースを高頻度に参照するアプリケーションがある場合、複数のスレーブサーバへ分散して参照することで、ひとつひとつのデータベースサーバへのアクセス負荷を抑えることができます。また、どれか1つのスレーブサーバに障害が発生してアクセス不可となっても、対象のスレーブサーバをシステムから切り離すだけでシステム復旧が可能です。アプリケーションの作りによっては、エラー時に次のスレーブサーバへ接続するよう自動リトライできますから、ダウンタイムを0にできます。

レプリケーションは、データベースのデータ復旧にも威力を発揮します。次ページの図でマスターサーバがディスク障害を起こしたとしても、スレーブサーバの1つをマスターサーバに昇格させたり、スレーブサーバのデータを構築しなおしたマスターサーバにコピ

ーすることで、障害発生直前までのデータを復旧させることができるわけです。

ただし、レプリケーションはバックアップとは違い、常にマスターと同様にデータを更新します。そのため誤削除などのオペレーションミスによるデータ喪失に対応することはできません。そのため、レプリケーションをしているからといってバックアップが不要ということはないので注意してください。

◆レプリケーションとは



注意事項

・このスクリプトにはMySQLサーバへの接続パスワードが書かれているため、ファイルのパーミッションに注意して、他のユーザからは不用意にファイルの中身を参照できないようにしてください。また、可能ならば接続ユーザは監視用の特別のユーザを作り、最低限の権限のみ与えるようにしましょう。万一スクリプトが他者から参照されてIDパスワードが漏洩しても、被害を最小限に抑えることができるためです。

関連項目

033　ファイルをバックアップする際にファイル名に日時を入れる
049　二重起動が可能な一時ファイル作成する

No. 122 MySQLのテーブルをCSV出力する

利用コマンド
mysql, date, tr

キーワード
MySQL, CSVファイル

いつ使うか
定期的にデータベースからSELECTした結果を、CSVファイルとして出力したいとき

実行例

```
$ ./mysql-csvout.sh
        csv_outputdirで指定したディレクトリにCSVファイルが出力される
```

スクリプト

```
#!/bin/sh

# データベース接続設定
DBHOST="192.168.11.5"
DBUSER="user1"
DBPASS="PASSWORD"
DBNAME="hamilton"

# mysql コマンドのフルパス
MYSQL="/usr/bin/mysql"

# CSV ファイルの出力パスと、レポート作成 SQL 文のファイル名を指定
csv_outputdir="/home/user1/output"
sqlfile="/home/user1/bin/select.sql"

# SQL ファイルの存在をチェック
if [ ! -f "$sqlfile" ];then
  echo "SQL ファイルが存在しません : $sqlfile" >&2
  exit 1
fi

# CSV ファイルの出力先ディレクトリをチェック
if [ ! -d "$csv_outputdir" ];then
  echo "CSV 出力先のディレクトリが存在しません : $csv_outputdir" >&2
  exit 1
fi

# 今日の日付を YYYYMMDD で取得
today=$(date '+%Y%m%d')
```

❶ ❷ ❸ ❹ ❺ ❻

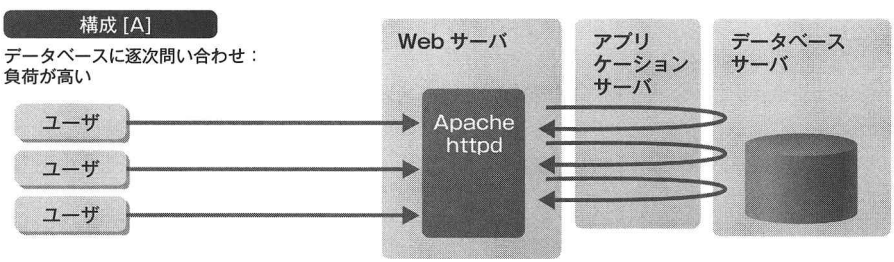
```
# CSV出力を行う。-Nでカラム名を表示しない。
# trコマンドでタブをカンマに置換する
$MYSQL -h "${DBHOST}" -u "${DBUSER}" -p"${DBPASS}" -D "${DBNAME}" -N ¥
  < "$sqlfile" | tr "¥t" "," > "${csv_outputdir}/data-${today}.csv"
                                                              ❼
```

解説

このスクリプトは、**MySQL**サーバにSELECT文を発行し、その結果をCSVファイルで出力するものです。MySQLサーバがリモートホストで稼働していても、このスクリプトを実行したマシン上に直接CSVファイルを作成することができます。なお、ここで発行するSQL文は、シェル変数sqlfileで指定したテキストファイルの中に記述していると仮定しています。

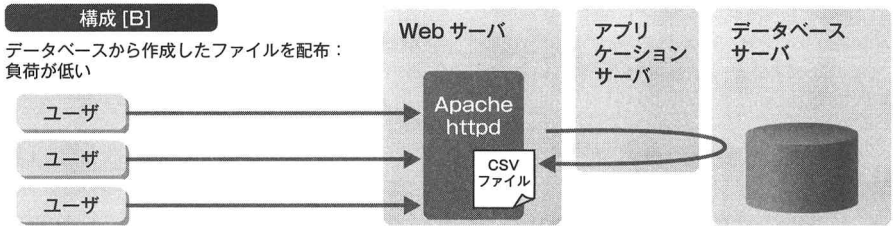
データベースの中身をSELECTした結果をレポートにして、それをWebサーバで配布してユーザPCで見てもらいたいというケースはよくあるでしょう。この際、リアルタイムにデータベースに問い合わせて結果を組み立てれば最新の情報を返却できますが、Webアクセスが集中するとデータベースへの負荷が高まってしまうという問題があります。特にWebサイトを外部に公開している場合、ニュースサイトに取り上げられるなど何かの拍子に急にWebアクセスが殺到するケースがあるため注意が必要です。

データベースからのレポート配布



対象のレポートが、ある1日ぶんの数値など日次データの場合には、毎日1回だけバッチ処理でCSVファイルを作り、そのファイルをユーザにダウンロードしてもらうようにすればデータベースサーバへの負荷を大幅に減らすことができます。このサンプル例は、このようなケースを想定しています。

毎回データベースにアクセスせずレポート配布



なおここで、シェル変数sqlfileで指定している、このサンプルで利用するSQL文はリスト1のようなものを想定しています。読者の環境にあわせて、このSQL文は適宜変更してください。

リスト1 サンプルのSQL文(select.sql)

```
SELECT id, score FROM userinfo ORDER BY id;
```

ではサンプル例を見ていきましょう。なお、ここでは❶で定義した設定でデータベースに正常に接続でき、❷のように**mysqlコマンド**がインストールされているものと仮定します。

❸では、CSVファイルの出力先ディレクトリ(シェル変数csv_outputdir)と、実行するSQL文を記述したファイル(シェル変数sqlfile)を定義しています。続いて❹と❺で、これらの存在チェックを行います。これらのファイルとディレクトリが存在しなければエラーとして、スクリプトはここで終了します。

❻では、dateコマンドを用いて現在の日付をYYYYMMDDのフォーマットで組み立てています。この**dateコマンド**でのフォーマット指定については、P.88で解説していますので詳しくはそちらを参照してください。この現在日付は、CSVファイル名を後で組み立てる際に利用します。

❼で、mysqlコマンドを利用してCSVファイルを出力します。ここではコマンド行が長くなるため行末に¥を置いて改行しています。ここで利用しているmysqlコマンドのオプションについては、次にまとめました。

スクリプトで使用しているmysqlコマンドのオプション

オプション	説明
-h	MySQLサーバのホスト名
-u	MySQLサーバに接続するためのユーザ名
-p	MySQLサーバに接続するためのパスワード
-D	接続するデータベース名
-N	カラム名を表示しない

❼では、mysqlコマンドに、SQL文が書かれたファイルを入力リダイレクトとして与えています。こうすると出力結果のカラムごとの区切りはタブで表示されてしまいます。そのため、パイプで渡した**trコマンド**で、タブを,(カンマ)に変換しています。具体的には、「tr "¥t" ","」としている部分です。これでSELECTした結果がCSV形式で得られますから、リダイレクトしてCSVファイルとして出力しています。

このようにして、MySQLでのSELECT結果を、シェルスクリプトでCSVファイルとして保存できます。cronに登録して、定期的なレポート作成などに利用すると便利でしょう。

注意事項

- MySQLで結果をCSVファイルとして出力したい場合には、次のような**INTO OUTFILE**命令を利用する手法がよく紹介されます。

```
SELECT id, score FROM userinfo
  INTO OUTFILE '/home/user1/output.csv'
  FIELDS TERMINATED BY ','
```

しかしMySQLのINTO OUTFILEは、mysqlコマンドを実行しているクライント上ではなく、データベースサーバ上に出力ファイルを作成するため、scpコマンドなどで、データベースサーバからネットワーク越しにファイルをローカルへコピーする必要があります。そのため本サンプルではINTO OUTFILEを利用せずに、SQL文の結果をリダイレクトして手元のマシンに直接保存しています。

- データ中に,(カンマ)を含む場合、区切り文字とデータ中のカンマの区別が付かなくなってしまいます。この場合にはSQL文のCONCAT関数を利用して、各カラムを"(ダブルクォート)でくくると、Microsoft Excelなどでは正常に処理できます。具体的には、SQL文を次のように修正すると各カラムの値をダブルクォートでくくることができます。

```
SELECT CONCAT('"',id,'"'), CONCAT('"',score,'"') FROM userinfo
```

ここで利用しているCONCATとはMySQLの文字列関数で、引数に指定した文字列を順に連結して1つの文字列として返します。SELECT文を発行する際に、選択したカラム値の前後を**CONCAT関数**を用いてダブルクォート記号で連結しておくことで、カラム値をダブルクォートでくくることができます。Microsoft Excelなどでは、値をダブルクォートでくくっておくと、データ中にカンマがあっても正しく区切りを扱えます。

関連項目

033 ファイルをバックアップする際にファイル名に日時を入れる

No. 123 ログ出力を監視し、ログに特定の文字列があれば警告する

利用コマンド
tail, read

キーワード
ログファイル, 監視, リアルタイム

いつ使うか
システムメンテナンス作業などの際に、監視対象のログファイルをリアルタイムに見やすく加工して出力したいとき

実行例

```
$ ./log-tailgrep.sh
！注意！ ファイルが見つかりません : [03ac2fsd.dat] File Not Found
！注意！ ファイルが見つかりません : [pxac2fsd.dat] File Not Found
！警告！ アプリケーション異常 : [6I7cht1npA] Application Error
```

リアルタイムにログを追って、注意メッセージを付けて出力する

スクリプト

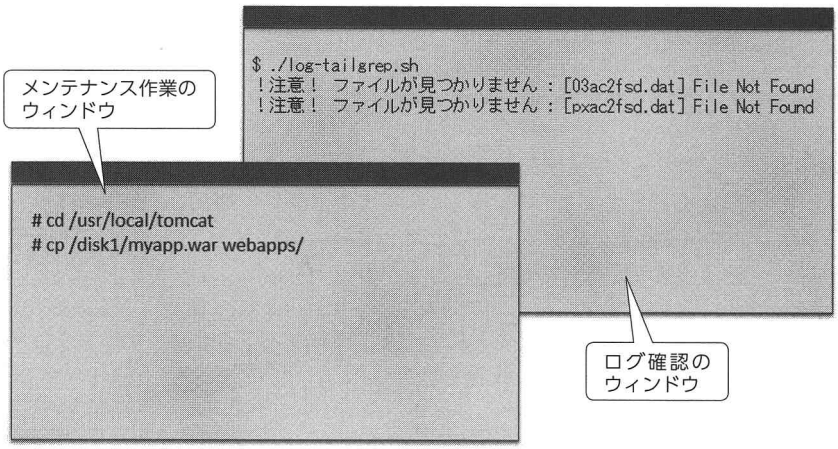
```
#!/bin/sh

# 監視対象のログファイル名を設定
logfile="/var/log/myapp/application.log" ―――――――――――――― ❶

# tailコマンドでログ監視：
#   * -F リアルタイムに監視
#   * -n 0 追記ぶんのみを対象とする
tail -F -n 0 "$logfile" |¥ ―――――――――――――――――――――――― ❷
while read line
do
  # ログにマッチする文字列があれば警告表示をする
  case "$line" in
    *"File Not Found"*)
      echo "！注意！ ファイルが見つかりません : $line"
      ;;
    *"Application Error"*)
      echo "！警告！ アプリケーション異常 : $line"
      ;;
  esac
done
```

❸

解説

このスクリプトは、常時追記されているログファイルに対して文字列を検索し、特定の文字列が含まれたログが出力された場合に、その内容をリアルタイムに表示するものです。端末ウィンドウで何かしらのシステムメンテナンス作業中に、別の端末ウィンドウ上でログの確認を行うツールとしての利用を想定しています。

スクリプトの利用例



ここで、メンテナンス対象のアプリケーションは、常時大量のログをapplication.logというログファイルに出力しているとします。この際、エラーメッセージが次のように素っ気ないものであるとします。

想定するapplication.logの出力

```
[03ac2fsd.dat] File Not Found
[pxac2fsd.dat] File Not Found
```

これはあまりに地味なので、エラーが出力された場合にすぐにわかるように、先頭に日本語で強調表示を付加したいというケースを想定します。

端末ウィンドウですぐわかるように、こう出力したい

```
!注意! ファイルが見つかりません : [03ac2fsd.dat] File Not Found
!注意! ファイルが見つかりません : [pxac2fsd.dat] File Not Found
```

サンプル例では、❶で監視対象のログファイルを指定しています。ここは読者の環境にあわせて、対象とするログファイルに修正してください。

❷で、ログ出力を監視してwhile文へとパイプでつないでいます。ここで利用している**tailコマンド**は、ファイルの末尾を表示するコマンドで、主なオプションは次のとおりです。

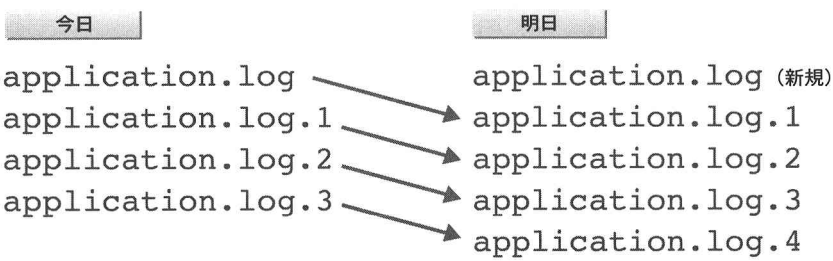
tailコマンドの主なオプション

オプション	意味
-f	ファイルの最後に達しても終了せず、追加されるデータを待つ
-F	-fオプションに加え、ファイルの名前変更やローテートも監視する
-n <数値>	ファイル末尾の<数値>行だけ表示する
-c <数値>	ファイル末尾の<数値>バイトだけ表示する

❷のtailコマンドでは、-Fオプションを利用しています。これにより、常時出力され続けるログファイルを、リアルタイムに追うことができます。一般にこの用途には小文字の-fオプションが利用されることが多いのですが、現在のtailコマンドでは、**ログローテート**にも対応している**-Fオプション**を利用したほうがよいでしょう。

ログローテートとは、アプリケーションのログファイルが肥大化しすぎないように、1日単位などでログファイルを切り替える機能です。ローテートの際は、次のようにファイル名の末尾に数値を付ける形式が一般的です。

ログローテート



この場合、application.logが翌日にはapplication.log.1とリネームされ、新規ファイルapplication.logが作成されます。つまり毎日、アプリケーションのログ出力が新しいファイルへと切り替わります。

ここで、tailコマンドの-fオプションを利用していると、変更前のファイルを読み続けるため、新しいapplication.logへの追記を検知することができません。-Fオプションを利用すると、(OSによって多少実装は違いますが) tailコマンドは定期的に該当ファイルを開きなおすため、ログローテートに対応することができます。

また❷では、-n 0として末尾ゼロ行を指定しています。これは、スクリプト実行時点ですでに存在するログファイルの中身は読み込まず、実行後に追記されたログのみを対象と

するための処理です。ここで-n 0を付けないと、tailコマンドのデフォルト動作である末尾10行をまず読み込みます。その末尾10行には過去に出力されたログがあるでしょうから、このようなリアルタイム監視の用途には不適切となるでしょう。

❷のtailコマンドの出力は、|(パイプ)でwhile文へとつないでいます。ここでは末尾に¥を置いて改行していますが、これは1行が長くなるのでwhile文を見やすくするためです。ここのwhile文では、**readコマンド**でシェル変数lineに追記されたログファイルを1行ずつ読み込みます。

❸のcase文が、追記ログへの検索部分です。シェル変数の値から特定の文字列を検索するにはいくつか手法がありますが、ここでは使いやすいcase文を利用して文字列を検索しています。まず"File Not Found"という文字列を含む場合、行頭に「！注意！ ファイルが見つかりません」と出力することにします。

❸のcase文でのマッチは、任意の文字列を表すワイルドカード*(アスタリスク)を使っています。なおここで、「"*File Not Found*"」のように*をダブルクォート記号の中に入れてしまうと、メタ文字ではなくアスタリスクそのものを意味してしまうためマッチしません。そのためここではアスタリスクをダブルクォート記号の外にだして、「*"File Not Found"*」としてマッチさせています。これでマッチしていた場合は、**echoコマンド**で先頭に「！注意！ ファイルが見つかりません」を付け、またその後ろに$lineを付けて元のログ内容も表示しています。

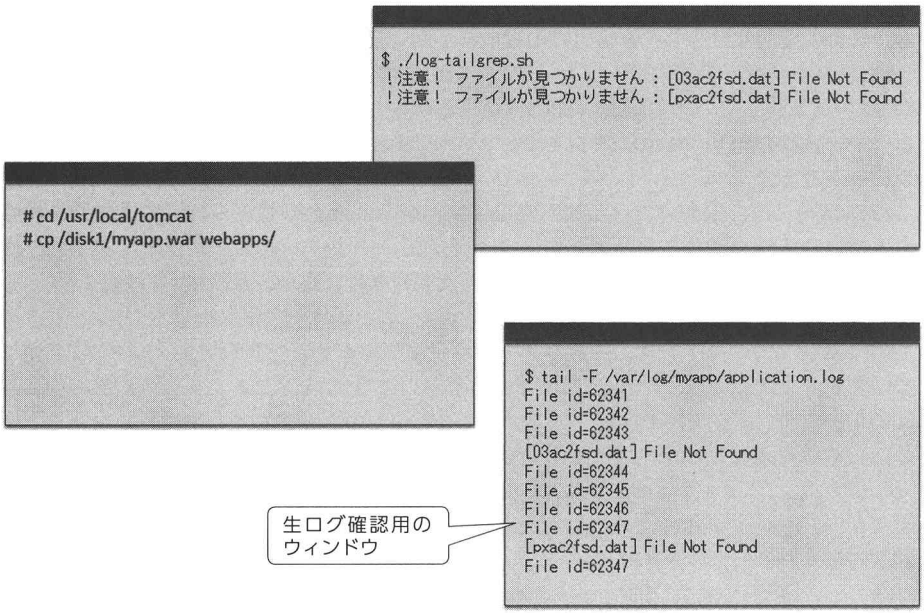
❸のcase文では同様に、"Application Error"という文字列も検索して、「！警告！」を表示するようにしています。他にもパターンを追記したければ、ここのcase文にいくつでも追加できます。

このようにして、指定したログファイルをリアルタイムに監視して、メンテナンス作業時などに見やすくすることができます。この他にも、何らかの読みにくいログを見やすいようリアルタイムにテキスト整形して表示したり、数値出力するプログラムのログをリアルタイムに計算して見せるといった使用法もあるでしょう。読者のニーズにあわせて適宜修正してみてください。

注意事項

・重要な作業時は、このスクリプトが何かしらのエラーを見逃す可能性もあるため、次のように「メイン作業の端末ウィンドウ」「このサンプル例のような、スクリプト加工したログ確認ウィンドウ」「生ログをtail -Fして表示しつづけるだけの端末ウィンドウ」と3つ開いておくのがよいでしょう。

●ウィンドウを3つ開いて使うとよい



lessコマンドの利用

常時追記されていくログファイルなどを監視する際には、tailコマンドの他に**lessコマンド**も利用できます。

lessコマンドはファイルを引数として実行し、指定されたテキストファイルの内容を閲覧するための、ページャと呼ばれるコマンドです。この際、ファイルの内容を表示中に、spaceキーで「次ページ」、[b]で「前ページ」へ移動することができます。

そして、[f]を押すことでlessコマンドはファイルの終端を読み続けるモードへと移行します。

●lessコマンドでFキー押下

```
[03ac2fsd.dat] File Not Found
[pxac2fsd.dat] File Not Found
Waiting for data... (interrupt to abort)
```

上図のように「Waiting for data...」と表示されるモードに移ると、lessコマンドは、tailコマンドの-fオプションと同様に、ファイルへ追記があれば自動的に読み込みます。

なお、lessコマンドのFモードから抜けるには、[Ctrl]+[C]を押します。lessコマンドの終了は[q]ですから、Fモードからそのまま終了するには[Ctrl]+[C]→[q]と順番に押します。

CHAPTER

10

bash

これまで本書では、互換性・移植性を考慮して/bin/shでのシェルスクリプトを紹介してきました。しかし最近のUNIXではbashがデフォルトでインストールされていることが多く、シェルスクリプトもbashで書かれることも多くなっています。
そのためこの章では、shではなくbashスクリプトとして、配列やブレース展開、算術式など、bash特有の機能を利用したサンプル例を紹介します。

No. 124 シェル変数を、整数値など属性付きで宣言する

利用コマンド
declare, curl

キーワード
bash, 変数, 属性, 型, 算術展開

いつ使うか
シェル変数を整数値として宣言し、算術展開を自動的に行わせたいとき

実行例

```
$ ./bash-declare.sh
        URLを自動生成してダウンロードが行われる
```

スクリプト

```
#!/bin/bash

url_template="http://www.example.org/download/img_%03d.jpg"

# カウンタ変数countを、整数値で宣言
declare -i count=0 ―――――――――――――❶

while [ $count -le 10 ]
do
  url=$(printf "$url_template" $count)
  curl -O "$url"

  # countを1増やす。exprコマンドなどは不要で、この式だけで1足せる
  count=count+1 ―――――――――――――❷
done
```

解説

このスクリプトは、連番ファイル名を生成して、Webページから順次ダウンロードします。P.275の例を、bashの整数値の変数を扱う機能で書きなおしたものです。

bashでは、シェル変数に対して属性値などを設定・参照する**declareコマンド**を利用することができます。このdeclareコマンドには、大きく分けて以下の2つの使い方があります。

1) 変数の属性を設定する
2) 現在定義されている変数・関数名を表示する

これらの2つの使い方についてそれぞれ説明します。

まず1) の変数の属性を設定する方法です。例えば次の例では、整数値のシェル変数であるcountを宣言するものです。

```
declare -i count=0
```

シェルスクリプトでは、基本的に変数の型はなく、すべてが文字列として扱われます。しかしdeclare文を利用することで、シェル変数に対して整数や配列などの属性を付加することができます。他のプログラミング言語で一般的に使われる変数宣言と似た概念ですから、この記法は比較的わかりやすいでしょう。

次に2) の現在定義されている変数・関数名を表示する方法です。例えば次の実行例は、現在スクリプト中で定義されている変数countの値を表示します。

シェル変数countを参照

```
$ declare -i count=2
$ declare -p count
declare -i count="2"
```

これは、スクリプトの途中で変数の状態を確認したいときなど、デバッグ用途によく使われる機能です。特に変数名が長いとき、いちいちechoコマンドを使って「echo "something_counter = $something_counter"」のように、変数確認のための文を挿入するよりも楽でしょう。

declareコマンドでよく使われるオプションは、属性設定と参照に分けて表にまとめました。なおbash 4.0以上では、変数を大文字の文字列専用として扱う「declare -u」など面白いオプションも追加されています。詳しくはbashのマニュアルを読んでみてください。

declareコマンドの属性設定オプション

オプション	意味
-a	変数を配列として扱う
-A	変数をハッシュ (連想配列) として扱う (bash 4.0以上)
-i	変数を整数として扱う
-r	変数を読み取り専用として扱う (readonlyコマンドと同じ)
-x	変数を環境変数として扱う (exportコマンドと同じ)

declareコマンドの参照オプション

オプション	意味
-f <関数名>	現在定義されているシェル関数とその内容を表示する。<関数名>を省略した場合は、定義されているすべてのシェル関数とその内容を列挙する
-F <関数名>	現在定義されているシェル関数の関数名のみを表示する。<関数名>を省略した場合は、定義されているすべてのシェル関数の関数名を列挙する
-p <変数名>	現在定義されている変数名とその値を属性付きで表示する。<変数名>を省略した場合は、定義されているすべての変数とその値を列挙する

declareコマンドの使い方がわかったところで、これを利用したサンプル例を見てみましょう。このサンプルは、P.275の例を利用したものであるため、スクリプトの動作の概要はそちらの説明を参照してください。前回はループカウンタをseqコマンドで生成しましたが、ここでは整数値として宣言したシェル変数countを利用します。

❶では、declareコマンドに、整数を宣言する**-iオプション**を利用しています。こうしてシェル変数countは整数値として扱われ、**算術展開**(Arithmetic Expansion)が適用されるようになります。

❷が、シェル変数countを整数値として定義したために利用可能となった記法です。通常のshスクリプトならばこの行は、次のようにexprコマンドで加算しないといけません。

```
count=$(expr $count + 1)
```

しかし❷では、一般的なプログラミング言語と同じような「<変数名>=<変数名>+1」という記法を利用できています。またこの際には、変数名に$を付ける必要もありません。ループ回数が大きい場合、exprコマンドをそのたびに実行すると処理時間が長くなってしまいますが、算術展開はbashの機能であるため毎回外部コマンドが実行されることもなく、処理を高速化できます。

なお❷の行は、C言語などと同じような記法で、次のように書くこともできます。

```
count+=1
```

この他に、算術展開は**$(())**という記法で書くこともできます。これら算術展開について詳しくは、P.370で説明していますのでそちらを参照してください。

注意事項

- Macなど一部の環境のshは機能拡張されているため、このサンプルであげた算術式をbashでなくとも扱うことができます。しかし基本的には算術式はbashの機能ですから、shで使うのは避けるべきです。
- FreeBSDではcurlは標準でインストールされていません。そのため次のように、代わりにfetchコマンドを利用するとよいでしょう。

```
fetch "$url"
```

COLUMN

シェルスクリプトはshで書くべきか？　bashで書くべきか？

シェルスクリプトを記述するには、移植性・互換性に配慮して、/bin/shが長いあいだ利用されてきました。しかしshは歴史の長いシェルである反面、数値計算に外部コマンドが必要であるなど欠点も多く、現代的なプログラミングがやりにくいのも事実です。そのため、bashでシェルスクリプトを書くことを好む人も昨今は増えてきているようです。

bashでシェルスクリプトを書く利点としては、単純にプログラミングがしやすいということもありますが、「移植性・互換性に配慮するならば、shよりbashのほうがむしろふさわしい」という意見もあります。

例えば同じLinuxでも、CentOSなどは/bin/shをbashで実装しているのに対し、Ubuntuではdashという原始的なシェルで実装しています。またSolarisのshは、Linuxのshに比べると動作にさまざまな差異があることがよく知られています。ですから、移植性・互換性を保ったつもりでsh用のシェルスクリプトを書いていても、環境ごとの「方言」に悩まされることは多々あります。一方、bashの実装はGNUによるものしかありませんから、マシンにbashが入っていればOSが何であれほぼ同じ動作となることが期待できます。つまり、bashスクリプトの移植性はとても高いのです。

このような状況をふまえて、筆者としては、現代のUNIX環境ならばshでもbashでも好きなほうで書けばよいと考えます（ただし、システムにbashがインストールされていなければshで書くしかありませんが）。しかしその際には、以下のようなルールは必要でしょう。

●bashの機能を利用したスクリプトのシバンは、#!/bin/shにしない

具体的には、算術展開や配列変数を利用している場合などです。Linux（CentOS）では/bin/shがbashを指しているため、bash特有の機能をshスクリプトとして記述しても問題なく動いてしまうことが多く、注意が必要です。bashスクリプトのシバンは、「#!/bin/bash（FreeBSDの場合は#!/usr/local/bin/bash）」と書くべきです。

●シェルスクリプトにこだわらない

最近では、システム運用のプログラムにPerlやPythonを使う事例が多く出てきました。シェルスクリプトにこだわらず、これらスクリプト言語の利用も検討してみてください。そもそも実現したい作業がPerlで書いたほうが楽ならば、shでもbashでもなく、Perlを使えばよいのです。

関連項目

099　連番のファイル名を持つURLを自動生成して、順にダウンロードする
126　足し算・かけ算などをシンプルに記述する

No. 125 forループをブレース展開で手軽に記述する

利用コマンド
ping, echo

キーワード
ブレース展開, bash

いつ使うか
IPアドレスリストなどをブレース展開の記法で手軽に作成したいとき

実行例

```
$ ./bash-brace.sh
[OK] Ping -> 192.168.2.1
[OK] Ping -> 192.168.2.2
[NG] Ping -> 192.168.2.3
[NG] Ping -> 192.168.2.4
[OK] Ping -> 192.168.2.5
```

スクリプト

```
#!/bin/bash

# bashのブレース展開{}でIPアドレスリストを作成する
for ipaddr in 192.168.2.{1..5} ―――❶
do
  ping -c 1 "$ipaddr" > /dev/null 2>&1

  if [ $? -eq 0 ]; then
    echo "[OK] Ping -> $ipaddr"
  else
    echo "[NG] Ping -> $ipaddr"
  fi
done
```

解説

このスクリプトは、**pingコマンド**を対象のIPアドレスに実行し、その結果をOK/NGで表示するものです。対象のIPアドレス作成には、bashの**ブレース展開機能**(Brace Expansion)を利用しています。

pingコマンドでのサーバ監視はP.323で紹介しましたので、サンプル例の概要はそちらを参照してください。P.323では1つのホストに3回pingコマンドを実行して丁寧に処理していましたが、ここではbashのブレース展開の機能にスポットを当てるため、スクリプトを簡略化してpingコマンド1回のみで応答をチェックしています。

bashのブレース展開とは、中カッコ{}で囲まれた文字列を展開する記法です。次にいくつかサンプルを例示しましたが、カンマ区切りで文字列を列挙する記法と、ピリオド2つで範囲を指定する記法の2種類があります。どちらも直感的なためわかりやすいでしょう。

bashのブレース展開のサンプル

```
$ echo index.{php,htm,html}  ―――― カンマ区切りで文字列を指定。リストを意味する
index.php index.htm index.html

$ echo 192.168.2.{1..5}  ―――― 数値をピリオド2つで範囲指定。「1から5」を意味する
192.168.2.1 192.168.2.2 192.168.2.3 192.168.2.4 192.168.2.5

$ echo host-{a..c}.example.com  ―― 文字列をピリオド2つで範囲指定。「aからc」を意味する
host-a.example.com host-b.example.com host-c.example.com
```

ブレース展開ではスペース区切りの文字列リストが得られるため、これをfor文のinに直接指定することができます。うまく使いこなすと大変に便利な記法ですので、ぜひ覚えておきましょう。

サンプルの❶では、IPアドレスのリスト作成にブレース展開を用いています。ここでは[192.168.2.1]から[192.168.2.5]までテストするため、第4オクテットにブレース展開を利用しました。実行時には、「192.168.2.1 192.168.2.2 192.168.2.3 192.168.2.4 192.168.2.5」と展開され、このIPアドレスに順番にpingコマンドが実行されます。

なおブレース展開は複数書くことができますので、例えばリスト1のように記述すると、192.168.0.0/16というネットワークに属するすべてのIPアドレスリストを作成できます。

リスト1 ブレース展開を複数書いた例

```
#!/bin/bash

for ipaddr in 192.168.{0..255}.{0..255}
do
  echo "$ipaddr"
done
```

このようにブレース展開を利用すると、さまざまなリストを簡単に作成できます。IPアドレスやファイル名の生成に利用してみるとよいでしょう。

関連項目

114 サーバのping監視を行う

No. 126 足し算・かけ算などをシンプルに記述する

利用コマンド
echo

キーワード
算術式, 算術展開, 算術評価

いつ使うか
exprコマンドを使わずに算術式を計算して、結果を得たいとき

実行例

```
$ ./bash-arithmetic.sh
        1.txtから100.txtまで、100個のファイルが作られる
```

スクリプト

```
#!/bin/bash

# ブレース展開で1から100までの数値リストを生成
for i in {1..100} ―――――――――――――――――――――――❶
do
  # 算術展開を利用し、ファイル名に3をかけ算した
  # 中身を持つテキストファイルを作成
  echo $((i * 3)) > ${i}.txt ―――――――――――――――――❷
done
```

解説

このスクリプトは、1から100までのファイル名を持つテキストファイルを作成し、そのファイルの内容はファイル名の数値に3をかけ算した値とするものです。何かしらのテストファイルを大量に作りたいときのツールとして使えるでしょう。

リストでは、まず❶でブレース展開を利用して1から100までの数値リストを作っています。このブレース展開については、P.368を参照してください。

❷では、bash特有の記法である**算術展開**(Arithmetic Expansion)を利用しています。この記法は**$((算術式))**として書かれ、カッコ内の算術式を評価し、その演算結果を数値として展開する記法です。ここで記述する算術式は、変数名の前に付ける$記号を省略できます。算術展開は一般的には、shシェルスクリプトにおいてexprコマンドで計算していた部分を置き換える目的に利用されます。

算術展開でよく利用される演算子を次にあげておきます。これらはJavaやC言語で利用される演算子とほぼ同じであるため、わかりやすいでしょう。

算術展開でよく利用される演算子(一部)

演算子	説明
+	加算
-	減算
*	乗算
/	除算(切り捨て)
%	剰余(割り算の余り)
**	べき乗
<<	左シフト(ビット演算)
>>	右シフト(ビット演算)

算術展開は、exprコマンドで計算するのに比べて、外部コマンドを利用しないことから速度面での大きな優位性があります。また変数名に$が不要であったり、乗算の演算子*にエスケープが必要ないなど、記述面でも便利です。

exprコマンドと算術展開の違い

```
result=$(expr $i ¥* $i)  ←exprコマンドでは、かけ算はエスケープが必要
result=$((i * i))        ←算術展開では、$不要・エスケープ不要で記述できる
```

なお算術展開と似た例に、**算術評価**(Arithmetic Evaluation)があります。これは算術式を評価し、その真偽値を返す記法です。その性質から、if文やwhile文の条件式としてtestコマンドの代わりに利用されることが多いでしょう。また、インクリメント/デクリメント演算子はそれ単体で成立し、演算結果を返す演算子ではないため、算術展開ではなく算術評価で利用されるのが一般的です。

リスト1は、算術評価を利用してwhileループを書いてみた例です。exprコマンドでの加算などがないため、シェルスクリプト特有の回りくどさがなくなっているのがわかります。

リスト1 算術評価を用いたループ処理

```
#!/bin/bash

# i=1からi=9までwhileループで処理
i=1
while ((i < 10))
do
  …何かの処理…

  ((i++))
done
```

このように、bashの算術展開・算術評価を利用するとさまざまな数値計算をシンプル

に記述することができます。またexprコマンドのオーバーヘッドもなくなるため、スクリプトの高速化も期待できるでしょう。

注意事項

- 算術展開では変数の頭に付ける$記号を省略できますが、引数を示す位置パラメータ($1, $2など)だけは$記号を省略できません。位置パラメータは$記号を省略すると、例えば「$1」は「1」となり、ただの数値と見分けが付かないためです。
- 算術評価には、letコマンドを利用する方法もあります。次のようにletコマンドの引数に算術式を渡すと、算術評価の記法(())と同様に、算術式が実行されてその真偽値が返されます。次の2行は同じ意味です。

```
((i++))
let 'i++'
```

- サンプル124で見たように、シェル変数をdecalare文の-iオプションで定義すると、対象の変数は$(())記法を使わなくても、算術展開を利用できます。

関連項目

124 シェル変数を、整数値など属性付きで宣言する
125 forループをブレース展開で手軽に記述する

No. 127 変数内の文字列を、n文字目からm文字取り出す

利用コマンド
read, echo

キーワード
文字列, 一部, 取り出し

いつ使うか
IDリストファイルから、指定したIDパターンのみを抽出したいとき

実行例

```
$ ./bash-substr.sh id.lst
AC38421021 0
AC98102495 1
```

IDの頭文字が"AC"のもののみ表示された

スクリプト

```
#!/bin/bash

# 指定されたIDファイルから、「$id $status」として1行ずつ
# readコマンドで読み込む
while read id status ―――――――――――――――――❶
do
  # シェル変数idの先頭2文字が"AC"かをチェックする
  if [ "${id:0:2}" = "AC" ]; then ┐
    echo "$id $status"           ├―❷
  fi                             ┘
done < "$1"
```

解説

このスクリプトは、IDリストファイル(id.lst)から「AC」で始まるIDを抜き出し、そのIDとステータスを表示するものです。ここで利用しているIDリストのファイルは、次のように「ID番号 ステータス」がスペース区切りで記述されていると仮定します。

リスト1 IDリストファイル(id.lst)の内容

```
AC38421021 0
HZ30281928 0
DP90023400 1
PX00031381 0
AC98102495 1
DP00281039 1
```

IDリストの処理は、シェルスクリプトがよく使われる分野の1つでしょう。特に、このサンプル例のようにIDの一部を取り出し、特定の文字列パターンを持つIDのみを抽出したい、というリスト処理は何かと利用シーンの多いケースです。

このサンプル例では、アンケート対象や抽選対象のID抽出を想定して、IDの先頭が「AC」で始まっているIDを抽出するシーンを考えます。このようなケースで必要な、「変数内の文字列に対して、指定したn文字目からm文字ぶん取り出す」という処理は、プログラミング言語によってはsubstrなどの関数で実装されています。しかしシェルスクリプトにはsubstr関数はないため、自前で処理しないといけません。ここではbashの**パラメータ展開**の機能を利用しています。

❶では、まずwhile文に対してコマンドライン引数で指定されたファイル($1)を入力リダイレクトしています。こうして**readコマンド**で1行ずつ読み込み、シェル変数idとstatusに順に代入して処理していきます。

❷のif文の条件式内で、シェル変数idに利用しているのが、n文字目からm文字ぶん取り出すというbashのパラメータ展開の記法です。この機能は次のように利用されます。

書式　パラメータ展開による文字列の取り出し

${変数名:オフセット:文字数}

　･･･「変数名」の、「オフセット」位置から後ろの「文字数」ぶん取り出す

${変数名:オフセット}

　･･･「変数名」の、「オフセット」位置から後ろの文字列をすべて取り出す

なおこの際、offsetは0から数えるため、「n文字目から取り出す」場合に実際に指定する数値は、nから1を引いた値となります。

次の実行例ではoffsetに4を指定していますから、シェル変数strの5文字目であるEから文字列が取り出されます。「${str:4:2}」と指定すると2文字取り出すのでEFが、「${str:4}」とlengthを省略すると4文字目から最後までのEFGHIJKが取り出されます。

●bashにおけるパラメータ展開は先頭位置が0となる

```
$ str="ABCDEFGHIJK"
$ echo ${str:4:2}
EF
$ echo ${str:4}
EFGHIJK
```

❷では、「${id:0:2}」と指定しています。これはIDの「1文字目から、2文字ぶん取り出す」ことを意味しますから、先頭2文字を取り出すことになります。これを"AC"と比較して、一致する場合にそのIDステータスをechoコマンドで表示しています。

もちろん、このような処理はgrepコマンドやsedコマンド、cutコマンドを使っても実現できます。しかしbashのパラメータ展開を利用すれば、そのような外部コマンドを使わずに実現でき、速度面でも有利であるため、ここで紹介してみました。

注意事項

・パラメータ展開の書式中、offsetやlengthは算術展開されるため、算術式をそのまま書くことができます。そのため次のように、何文字目から取り出すかをシェル変数で動的に指定できます。なお算術展開については、P.370を参照してください。

```
num=1

# 取り出すオフセット値を num+2 に指定
echo ${id:num+2:2}
```

また上記の例を応用すると、変数の末尾1文字のみを取り出す処理を、以下のように書くことができます。

```
echo ${id:${#id}-1}
```

ここで${#id}とは、シェル変数idの値の文字列長です。

関連項目

126 足し算・かけ算などをシンプルに記述する

No. 128 変数内の文字列の一部を置換する

利用コマンド
echo

キーワード
パラメータ展開, 文字列, 置換, sed

いつ使うか
変数内の文字列を、sedコマンドを使わずに置換したいとき

実行例

```
$ ./bash-where.sh perl
/opt/local/bin/perl
/usr/bin/perl
```

スクリプト

```
#!/bin/bash

# 探査コマンドを取得
command="$1" ―――――――――――――――――――――――――――――――――――❶

# 引数チェック
if [ -z "$command" ]; then
  echo "エラー：調べるコマンドを指定してください" >&2
  exit 1
fi ―――――――――――――――――――――――――――――――――――――――――――❷

# 環境変数$PATHのコロンをスペースに置換し、for文のループ対象の
# リストとする
for dir in ${PATH//:/ } ―――――――――――――――――――――――――――――――❸
do
  if [ -f "${dir}/${command}" ]; then
    echo "${dir}/${command}"
  fi
done
```

解説

このスクリプトは、コマンドライン引数で指定されたコマンドが環境変数のパス内にあるかどうかを探索し、存在する場合はそのフルパスを列挙するものです。実行例では、perlコマンドが/opt/local/binと/usr/binの2カ所に見つかったとして表示しています。

このサンプルでは、tcshというシェルのwhereコマンドを模した動作をするスクリプトを作っています。tcshとは、bashとは違う系列のCシェルの一種で、一昔前はBSD系の

UNIXでよく使われていました。このtcshには、あるコマンドがパス内のどこに存在するかをすべて列挙するwhereというコマンドがあります

bashにはwhereコマンドがないため、このサンプルで作成してみることにします。例えばPerlがシステムに違うパスで複数インストールされている場合、その状態確認のために使うツールとして便利かもしれません。

リストの❶では、まずコマンドライン引数で指定されたコマンド名を、位置パラメータ$1からシェル変数commandに代入します。❷でその引数の中身をチェックし、空文字列かどうかをtestコマンドの-zオプションでチェックしています。空文字列ならばコマンドが指定されていませんから、エラー表示してexitコマンドで終了しています。

❸で、bashの**パラメータ展開**の機能を使って:(コロン)をスペースに置換しています。このような文字列置換は一般的にはsedコマンドが使われますが、bashでは標準機能であるパラメータ展開で置換できるため、外部コマンドであるsedを使う必要がありません。

bashのパラメータ展開による文字列置換は、次のような書式で利用します。

書式 パラメータ展開による文字列置換

(1) ${変数名/パターン/置換文字列}

・・・「変数名」の、最初に一致した「パターン」のみ、「置換文字列」で置換する。sedコマンドの 's/パターン/置換文字列/' に相当

(2) ${変数名//パターン/置換文字列}

・・・「変数名」の、すべての一致する「パターン」を「置換文字列」で弛緩する。sedコマンドの 's/パターン/置換文字列/g' に相当

「パターン」の指定には、ワイルドカードである*(任意の文字列)や?(任意の1文字)、文字クラス指定[...]なども利用することができます。

❸で利用する変数PATHは、コマンドの探索パスを指定する環境変数で、次のようにコロン区切りで指定されています。

環境変数PATHの設定形式

```
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin
```

ここでではこのコロンをスペースに置換することで、for文のリストとして各ディレクトリをシェル変数dirに代入しています。それぞれのパス内にコマンドライン引数で指定されたコマンドが存在しているかを、testコマンドのファイルテスト演算子-fを利用してチェックし、ファイルがあればechoコマンドで表示しています。

このようにして、パスが通っているディレクトリのどこにコマンドがあるか、一覧表示できます。同一コマンドが複数のパスにインストールされていないかのチェックなどに使

えるでしょう。

注意事項

- このスクリプトでは、for文でスペースを区切りとして用いているため、環境変数PATHにスペースを含むディレクトリを指定している場合には正しく動作しません。
- パラメータ展開が使えないshスクリプトでは、変数内の文字列を置換する際には、次のようにechoコマンドで変数の値を出力してから、それをsedコマンドで処理する必要があります。

```
echo $PATH | sed 's/:/ /g'
```

- 文字列置換の際に、文字列の先頭のみにマッチする/#という記法も利用できます。これは正規表現では^（行頭）で表されるパターンです。例えば次のように、先頭に余計なスペースが入っている文字列と入っていない文字列がある場合、/#で置換すると先頭にスペースがあるほうは置換されません。

```
# 先頭に余計なスペースが入っている
str1="  http://www.example.com/"
# 先頭にスペースがない
str2="http://www.example.com/"

echo ${str1/#http:/hxxp:}
echo ${str2/#http:/hxxp:}
```

上記の出力結果は次のようになり、1つ目のシェル変数str1は先頭にスペースが入っているためマッチしていないことがわかります。

```
http://www.example.com/
hxxp://www.example.com/
```

なお、これとは逆に、末尾にマッチさせたい場合は/%記法を用います。

No. 129 中間ファイルを作らずにコマンドの出力をファイルのように扱う

利用コマンド
comm

キーワード
プロセス置換, 名前付きパイプ, FIFO

いつ使うか
コマンドの出力結果をそのままファイルのように扱い、中間ファイルを作らないようにしたいとき

実行例

```
$ ./process-subst.sh
                app.log.20140201
                app.log.20140202
                app.log.20140203
                app.log.20140204
        app.log.20140205
                myapp
obj.conf
```

obj.confはdir1のみにあり、app.log.20140205はdir2のみにある

スクリプト

```
#!/bin/bash

# 比較する2つのディレクトリの定義
dir1="/var/tmp/backup1"
dir2="/var/tmp/backup2"

# commコマンドで出力を比較。中間ファイルを作らなくても
# プロセス置換で処理できる
comm <(ls "$dir1") <(ls "$dir2")
```

❶ ❷

解説

このスクリプトは、2つのディレクトリ内のファイル一覧を比較して、その差異を表示するものです。サンプル044で紹介したcommコマンドを、中間ファイルを作らずにbashのプロセス置換(Process Substitution)の記法で処理しています。

bashの**プロセス置換**とは、コマンドの入力や出力を、FIFO(名前付きパイプ)やファイ

ル記述子を示す/dev/fd配下のデバイスファイルに接続して実行する機能です。コマンドの出力結果を、ファイルとして別コマンドに渡すには、次のような書式を使います。

書式　プロセス置換の使い方

```
コマンド1 <(コマンド2)
```

一般的な利用の仕方としては、ファイルを対象とするコマンドを実行する際に、中間ファイルを作らずに処理する用途が多いでしょう。例えば次の例では、lsコマンドの実行結果をファイルとみなして、commandAが実行されます。

```
commandA <(ls -alF)
```

サンプル例では、commコマンドで比較する対象リストファイルを、lsコマンドの結果とするために、プロセス置換を用いています。まず❶で、比較する2つのディレクトリを定義しています。この2つのディレクトリ内のファイルリストを比較するというケースを想定しています。

❷で、プロセス置換を用いて2つのlsコマンドの結果を比較しています。commコマンドは本来は、次のように2つのファイルを引数にとります。

```
comm a.txt b.txt
```

しかし❷では、2つのファイルをプロセス置換として指定することで、コマンドの結果をそのまま利用することができます。

プロセス置換を使わない場合には、次のように2つのlsコマンドの結果をいったん一時ファイルに出力しておいて、それを比較するという手間がかかります。プロセス置換を使うことで、このような一時ファイルの作成という面倒な作業をなくすことができたということです。

```
ls "$dir1" > dir1.tmp
ls "$dir2" > dir2.tmp
comm dir1.tmp dir2.tmp
```

このようにしてcommコマンドの結果として、「ディレクトリ$dir1のみにあるファイル」「ディレクトリ$dir2のみにあるファイル」「両方にあるファイル」を分けて出力することができます。プロセス置換はこの他にも、ファイルを対象とするコマンドで応用できる記法です。

注意事項

・プロセス置換の記法「<(コマンド)」は、一見すると「入力リダイレクトとサブシェル」のように見えますが、この理解は誤りです。あくまで、「<(コマンド)」で1セットの記法です。例えばリストの❷を、次のようにしてカッコの前にスペースを入れると、エラーとなりますので注意してください。

```
comm < (ls "$dir1") < (ls "$dir2")
```

・プロセス置換には、この例で見たような入力をつなぐ処理とは逆に、出力をつなぐ「>(コマンド)」という記法もあります。これは処理結果を標準出力には出さずにファイルに直接書き込むようなプログラムで、パイプ処理したいときに利用できます。例えばmy-calcコマンドが、直接結果を指定されたファイルへ出力する仕様のコマンドであるとすると、次のように記述することで、結果をファイルではなく標準出力に行番号付きで表示することができます。

```
my-calc >(cat -n)
```

・FreeBSDのバージョンによっては、プロセス置換が正しく動作しない場合があるようです。その際にはportsからbashの最新版をインストールしてみてください。

関連項目

044 2つのディレクトリ内を比較し、どちらか片方だけに存在するファイルを表示する

No. 130 パイプ処理で各コマンドの終了ステータスを調べる

利用コマンド
echo

キーワード
パイプ, 終了ステータス

いつ使うか
パイプ処理の途中にあるコマンドの終了ステータスをチェックし、コマンドの成功・失敗を判断したいとき

実行例

```
$ ./pipestatus.sh
[ERROR] sort-data.sh は失敗しました
```

スクリプト

```
#!/bin/bash

# 以下のような処理を行うケースを想定する。
#   script.sh : データ出力
#   sort-data.sh : データをソートする
#   calc.sh : 出力データの計算をする
./script.sh | ./sort-data.sh | ./calc.sh > output.txt ―――❶

# 別のコマンドを実行するとPIPESTATUSの値が失われるため、
# 結果をコピーしておく
pipe_status=("${PIPESTATUS[@]}") ―――――――――――――――――――❷

# パイプ処理の途中のコマンドの成功失敗チェック。
# sort-data.shの終了ステータスが非ゼロかを確認する
if [ "${pipe_status[1]}" -ne 0 ]; then
  echo "[ERROR] sort-data.sh は失敗しました" >&2
fi
```

❸

解説

このスクリプトは、パイプ処理の途中でエラーが起きたかどうかをチェックし、エラーが発生していた場合はエラーメッセージを出力するものです。

ここでは、script.sh → sort-data.sh → calc.sh という3つのスクリプトを順に実行して何らかのデータ処理を行い、その結果をoutput.txtというファイルに出力するケースを想定します。ここで、sort-data.shは失敗することもありますが、後続のcalc.shでの計算にソートは必須ではないため、エラーは起きてもかまわず実行するものと仮定します。ただし、ソートが失敗した際には終了ステータスが非ゼロとなるため、これを検知してエラ

ーメッセージだけは出力しておきたい、というケースを考えます。

コマンドの終了ステータス$?を利用するサンプル例は本書でも多く取り上げていますが、パイプ処理をしている場合には、途中のコマンドの終了ステータスを$?で取得することはできません。変数$?には**パイプラインの最後に実行されるコマンドの終了ステータスしか代入されない**ため、例えば❶では最後のcalc.shの終了ステータスしか取得できないのです。

しかし、bashの組み込み変数である**PIPESTATUS**を利用すると、直前のパイプライン処理のすべての終了ステータスを取得することができます。PIPESTATUSは配列変数となっており、その0番目にはパイプの1つ目の終了ステータスが、1番目にはパイプの2つ目の終了ステータスが入っています。

本サンプルにおける特殊変数PIPESTATUSの値

記述例	説明
${PIPESTATUS[0]}	パイプ処理の1つ目 (script.sh) の終了ステータス
${PIPESTATUS[1]}	パイプ処理の2つ目 (sort-data.sh) の終了ステータス
${PIPESTATUS[2]}	パイプ処理の3つ目 (calc.sh) の終了ステータス

❷の${PIPESTATUS[@]}で利用している配列の添え字@は、配列すべてを意味します。全体をカッコ()でくくることで、配列全体をシェル変数にコピーしています。

❸で、sort-data.shの終了ステータスを${PIPESTATUS[1]}から取得して、testコマンドの、等しくないかどうかを調べる-ne演算子で0と比較しています。終了ステータスが0ではない場合には、エラーメッセージを表示しています。

このようにして、パイプ処理の途中のコマンドの終了ステータスを取得することができます。細かいエラー処理をしたいときに役に立つでしょう。

PIPESTATUS利用の注意点

変数PIPESTATUSの利用には注意すべき点があります。それは、何かコマンドを実行するたびに、変数PIPESTATUSは常に更新されてしまうということです。例えばリスト1は、パイプ処理した3つのコマンドの終了ステータスをechoコマンドで確認しようとした(つもりの)スクリプトです。しかしこのスクリプトは、誤った書き方をしています。

リスト1 PIPESTATUSの誤った使用例(list2.sh)

```
#!/bin/bash

./script.sh | ./sort-data.sh | ./calc.sh > output.txt

echo ${PIPESTATUS[0]}
echo ${PIPESTATUS[1]} ―――――――――――――――Ⓐ
echo ${PIPESTATUS[2]} ―――――――――――――――Ⓑ
```

リスト1を実行してみると、Ⓐ とⒷのechoコマンドが何も表示されません。

リスト1の実行例

```
$ ./list2.sh
0
```

これは、パイプを使っていないコマンド処理でも、変数PIPESTATUSは0個のパイプ処理があるとみなして、常に更新されてしまうからです。よって、Ⓐでは直前のechoコマンドの2番目のパイプ処理の終了ステータスを取得しようとして、そのような処理が存在しないため空文字列を出力しています。Ⓑも同様です。このように、echoコマンドなど何かコマンドを実行するたびに、PIPESTATUSは上書き更新されてしまいます。そのため本サンプルでは、PIPESTATUSをいったんシェル変数pipe_statusに丸ごとコピーしています。

注意事項

・シェル変数PIPESTATUSは、shには存在しないbash特有のシェル変数です。この他にもbash特有のさまざまなシェル変数があります。その中からシェルスクリプトを書く際に便利そうなものの一部を、次にあげておきました。

bash特有の便利なシェル変数

シェル変数名	意味
BASH	現在実行しているbashを起動したときのフルパス
DIRSTACK	現在のディレクトリスタックの内容(配列変数)
SHELLOPTS	現在有効になっているシェルのオプション(コロン区切り)
SECONDS	シェルが起動されてからの秒数
HOSTNAME	ホスト名
UID	現在シェルを実行しているユーザのユーザID
GROUPS	現在のユーザがメンバになっているグループのリスト(配列変数)

それぞれの詳細や、この他のbash特有のシェル変数については、man bashとしてbashのマニュアルを読んでみてください。

No. 131 ユーザに簡易メニューを表示して選択してもらう

利用コマンド
select, case

キーワード
メニュー, 選択

いつ使うか
簡単なメニュー表示をして、ユーザに操作させたいとき

実行例

```
$ ./bash-select.sh
1) list file
2) current directory
3) exit
Menu: 2
/home/user1/bin

Menu: 3 ―――― ユーザが3を入力するとスクリプトが終了する
```

スクリプト

```
#!/bin/bash

# メニュープロンプト文の定義
PS3='Menu: ' ――――――――――――――――――――――――――――――――――❶

# メニュー表示の定義。メニューの各項目はinへのリストとして指定する。
# $itemには選択されたリストの文字列が、$REPLYには入力された数値が代入される
select item in "list file" "current directory" "exit"
do
  case "$REPLY" in
    1)
      ls
      ;;
    2)
      pwd
      ;;
    3)
      exit              # 選択肢にexitコマンドがないとメニューから抜けられない
      ;;
    *)
      echo "Error: Unknown Command"  ――――――――――――――――――――――❷
      ;;
  esac
```

```
  echo
done
```

解説

このスクリプトは、メニュー表示をするものです。メニュー1番の"list file"を選ぶとlsコマンドを、2番の"current directory"を選ぶとpwdコマンドを、3番の"exit"を選ぶとexitコマンドを実行します。このようなメニュー表示のスクリプトはP.18で作りましたが、ここではbashの機能であるselect文を利用しています。

bashのselect文とは簡単にメニューを作るための機能で、次のように利用します。

書式 select文の基本的使用法

```
PS3=プロンプト文
select <変数名> in <リスト>
do
   ....(コマンド)
done
```

<リスト>で指定した文字列を元にbashがメニューを組み立てて、自動的に番号を振って表示してくれます。select文の「in <リスト>」の部分は省略可能で、省略すると位置パラメータ$@、つまりコマンドライン引数が指定されたものとみなされます。

また**PS3**とは、select文が利用するbashのシェル変数で、この文字列がメニューへのプロンプトとして表示されます。

select文では、ユーザが選択したリストの値が、<変数名>に代入されます。またこの際、ユーザが入力した数値は同時にシェル変数REPLYに代入されます。メニューとしてcase文を利用して分岐させるならば、このシェル変数REPLYを利用すると便利でしょう。サンプル例でも、シェル変数REPLYを利用して選択されたメニューを取得しています。

サンプルスクリプトでは、まず❶でシェル変数PS3にプロンプト文を代入しています。ここで設定した値が、メニュー表示の際の問いかけ文として表示されます。

❷では、select文でメニューを定義しています。ここでは"list file" "current directory" "exit"の3つの文をリストとして与えています。実行例で見るように、これらに順に1)、2)、3)と数値を付けて表示するのはbashが自動的に行ってくれますから、スクリプト中に記載する必要はありません。ユーザが入力した数値は、select文によってシェル変数REPLYに代入されていますから、これをcase文で分岐して各処理を実行しています。

このようにして、簡単にメニューを作成することができます。操作に不慣れな初心者が利用するスクリプトなどで、対話型のメニューを利用したいときに役立つでしょう。

No. 132 整数値の乱数を得る

利用コマンド
nc, echo, sleep

キーワード
乱数, 整数

いつ使うか
外部コマンドを使わず、シェルスクリプトだけで乱数を取得したいとき

実行例

```
$ ./bash-random.sh
Connection to 192.168.2.1 80 port [tcp/http] succeeded!
Wait: 4 sec.
Connection to 192.168.2.1 80 port [tcp/http] succeeded!
```

スクリプト

```
#!/bin/bash

# テスト通信先の定義
ipaddr="192.168.2.1"
port=80

# 1から10までの整数値の乱数を、RANDOM変数から取得する
waittime=$((RANDOM % 10 + 1)) ❷

# テストコマンドを、ウェイトを入れて2回実行する
nc -w 5 -zv $ipaddr $port
echo "Wait: $waittime sec."
sleep $waittime
nc -w 5 -zv $ipaddr $port
```

❶ ❸

解説

このスクリプトは、ncコマンドによるネットワークの通信テストを2回行う際、合間のウェイト秒として乱数を用いるものです。乱数はbashのシェル変数RANDOMを利用しています。このスクリプトは、乱数を取得する部分にbashの機能を用いたこと以外はP.268のサンプルと同じものですから、スクリプトの詳しい動きについてはそちらを参照してください。

さて、シェルスクリプトを書いていると、テストのためのウェイト秒数やゲームなどでの「サイコロ」など、乱数が必要な機会がときおりあります。しかしsh自体には乱数を生成する機能はないため、/dev/urandomなどを用いて乱数を組み立てる必要がありました。

しかしbashにはデフォルトで、乱数を生成してくれる機能があります。それがシェル変数**RANDOM**です。

bashのシェル変数RANDOMは、参照するたびに0から32767までの整数値の乱数を返します。これを利用して、1からnまでの乱数を得たい場合には、次のように算術展開を利用して記述します。なおbashの算術展開については、P.370で説明していますので詳しくはそちらを参照してください。

```
$((RANDOM % n + 1))
```

上記では算術式の剰余（割り算の余り）を意味する演算子%を利用しています。例えばnが10ならば、得られる剰余は0から9ですから、それに1を足せば1から10までの乱数が得られるわけです。これはよく使われるイディオムのようなものですから、丸ごと覚えておくとよいでしょう。

❶では、まず通信テストを行う対象ホストのIPアドレスとポート番号を指定します。続いて❷で、テスト間隔のウェイト秒数を乱数で取得します。ここではウェイト秒数に1から10の乱数を得たいため、RANDOM変数の10の剰余に1を足しています。得られた乱数を、シェル変数waittimeに代入しています。

❸で、**ncコマンド**で通信テストを2回行い、合間に先ほど❷で取得した乱数のウェイト秒を入れています。ここでのncコマンドのオプションなどは、P.162を参照してください。

このようにして、bashでは簡単に乱数値を得ることができます。覚えておくと、何かの機会に役に立つかもしれません。

関連項目

097 シェルスクリプトの一部にPerlやRubyを使う
126 足し算・かけ算などをシンプルに記述する

APPENDIX

追加情報

01 端末（ターミナル）とは

端末(ターミナル)

UNIX環境では、端末とはユーザが操作するシェルへの入出力インタフェースを指します。現在では、端末といえばほぼ「端末ウィンドウ」を指すと思って間違いないでしょう。これは端末エミュレータとも呼ばれるソフトウェアで、次のような種類があります。この本を読んでいる読者も、これらのどれかを使っていることでしょう。

主な端末エミュレータ

環境	ソフトウェア名称
Windows	TeraTerm, PuTTY, Poderosa
Linux,FreeBSD	xterm, kterm, gnome-terminal, Konsole
Mac	Terminal.app, iTerm2

しかし本来、端末とはこのようなソフトウェアのみを指すのではなく、もっと広い意味を持ちます。例えばシリアルポートを介して利用するテレタイプ(電動タイプライター)も端末であり、これはいまもUNIXのttyデバイスという名称に名前を残しています。また読者の中には、データセンターなどでダム端末を利用した経験のある方もいることでしょう。

サンプル004などでは端末操作のためにsttyコマンドを利用していますが、このsttyコマンドにはボーレート(変調回数)やパリティの設定を行う機能があります。これは、シリアルポートにダム端末などをつなぐ際に、各種設定をするための機能が残っているものです。

仮想端末

端末エミュレータでUNIXにログインすると、それは仮想端末を利用していることになります。次のように**ttyコマンド**を実行すると、現在利用している仮想端末のデバイス名を確認することができます。

仮想端末のデバイス名を表示

```
$ tty
/dev/pts/0
```

一般に、端末デバイス名はttyが物理ディスプレイとキーボード、ptsは仮想端末を指し

ます(Macの場合はデバイス名の割り当てが多少違います)。

ここで1つ、簡単な実験をしてみましょう。端末ウィンドウを2つ開いて、それぞれ同じUNIXマシンにログインしてください。そして片方の端末ウィンドウでttyコマンドを実行し、割り当てられている仮想端末のデバイス名を確認します。ここでは、「/dev/pts/1」が割り当てられていたと仮定します。

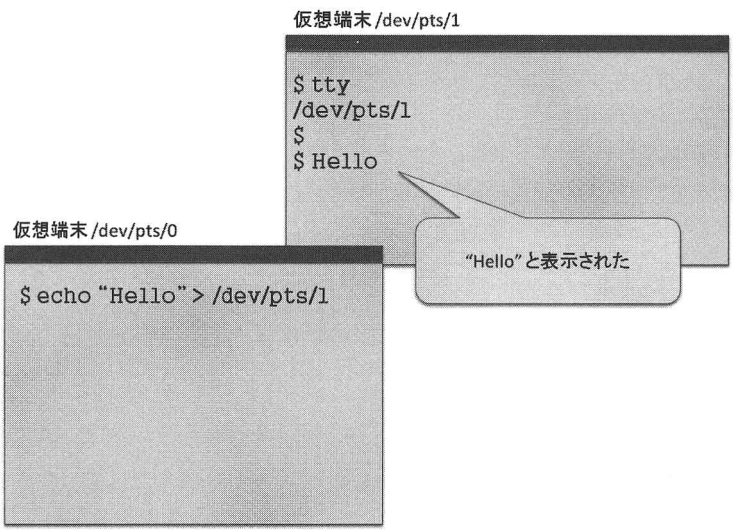
続いて、もう片方の端末ウィンドウから、次のように"Hello"という文字列を表示するechoコマンドを実行します。ここで、先ほど確認した仮想端末のデバイスファイルに出力リダイレクトを行います。

仮想端末にechoコマンドの出力をリダイレクト

```
$ echo "Hello" > /dev/pts/1
```

すると次のように、もう1つのウィンドウに"Hello"が表示されます。

仮想端末とデバイスファイル



このようにUNIXでは、デバイスファイルに端末を割り当てています。この実験で、端末とその動きが理解できたのではないでしょうか。

02 UNIXコマンドのオプションについて

サンプル001でgetoptsコマンドによるオプションの利用例を紹介しました。ここではlsコマンドを例にあげて、UNIXで使われるオプション指定についての基礎知識を解説します。

オプションの指定方法

UNIXの多くのコマンドは、オプションを指定することができます。オプションは-（ハイフン）の後ろに、1文字のアルファベットもしくは数値を付加して指定します。

```
ls -l
```

上記は、lsコマンドに、ファイルの詳細情報を表示するオプションである-lを指定したものです。lsコマンドは何もオプションを付けないとファイル名しか表示しませんが、このように-lオプションを付けるとファイルのパーミッション、所有者、タイムスタンプなどの詳細情報を表示します。

オプションを複数指定するには、それらを並べて書きます。この際、ひとつひとつにハイフンを付けて分けて書いてもよいですし（❶）、ハイフン1つに複数のオプションの文字をまとめて書いてもかまいません（❷）。

```
ls -a -l ―――― ❶
ls -al ―――― ❷
```

ロングオプション

オプションは、伝統的にはハイフン1つにアルファベット1つで指定されます。しかし最近では、ハイフン2つに英単語を並べて指定するロングオプションも用いられています。次は、Linuxでlsコマンドの結果をカラー表示するオプション--colorを用いた例です。

```
ls --color
```

このロングオプションは、GNUの提供するコマンド群に多く見られることからGNU形式とも呼ばれます。ロングオプションは英単語で構成されるため、ひと目で見て効果がわかりやすいのが利点です。

しかしFreeBSDなどのBSD系UNIX環境では、POSIX形式である1文字オプションにこ

だわる傾向が強く、このロングオプションはあまり用いられません。例えばFreeBSDでは、lsコマンドをカラー表示するには--colorオプションは使えず、代わりに-Gオプションを用います。

なおロングオプションは、他のオプションと区別できる範囲で省略して書けます。例えばlsコマンドの--colorオプションは、「ls --col」とだけ書いても正しく認識されます。

オプションの引数指定

オプションには、引数を指定するものがあります。例えばheadコマンドの表示行数を指定する-nオプション(LinuxではGNU形式の--linesオプションも利用可能)では、表示する行数を数値で指定します。引数指定の際には、1文字オプションでは後ろにスペースを入れても入れなくてもかまいません。またGNU形式ではスペースではなく=を使ってもかまいません。以下4つはすべて同じ意味で、「先頭2行を表示」します。

```
head -n 2 <filename>
head -n2 <filename>
head --lines 2 <filename>
head --lines=2 <filename>
```

ハイフンで始まるファイルを引数にとる

あまり見られないケースですが、ファイル名がハイフンで始まるファイルを扱いたい場合があります。例えば次の例では「-sample.txt」というファイル名を引数にとりたいのですが、このファイル名がオプションだと判断されて、エラーになっています。

```
ls -sample.txt
```

このような場合は、オプションとしてハイフン2つ--を書くと、それ以降にオプションはないという打ち切りを意味します。そのため--の後ろにハイフンを含むファイル名を書けば、引数を正しく扱うことができます。

```
ls -- -sample.txt
```

03 シェルスクリプトの変数名

大文字と小文字

シェルスクリプトでは、変数名に大文字小文字どちらも用いることができます。

特に文法的な決まりではありませんが、一般的には他のプログラミング言語と同じく、定数は大文字で、変数は小文字で書いたほうがよいでしょう。また、環境変数は大文字で書くのが普通です。

大文字と小文字の使い分け

```
# 定義ファイル名は定数なので大文字
CONFIGFILE="myapp.conf"

# ループカウンタは変数なので小文字
i=0

# 環境変数は大文字
MY_TMPDIR="/var/tmp"
export MY_TMPDIR
```

ただし、すべての変数名を大文字で書き、小文字は一切使わないという流儀の人もいます。プロジェクトによってはコーディング規約などですべてを大文字変数とすることもありますから、そのようなスクリプトを見ても驚かないようにしてください。

また、シェルスクリプトで変数名や関数名が長くなる際には、キャメルケースはあまり使われず、スネークケースが利用されることが多いようです。

ここでキャメルケースとは、複数単語からなる場合にそれぞれの単語の先頭を大文字にする記法で、特にJavaでよく使われる命名法です。例えばJavaのFileクラスで新規ファイル作成をするメソッド名は、"createNewFile"です。一方、スネークケースとはそれぞれの単語を_(アンダーバー)でつなぐ記法で、古くからさまざまな言語で使われています。

キャメルケース：sampleDataIndex　(Javaでよく使われる)
スネークケース：sample_data_index (シェルスクリプトでよく使われる)

ちなみにキャメルケースのキャメル(Camel)とは、ラクダのことです。おそらく変数名の大文字小文字が入り交じるデコボコが、ラクダのこぶのように見えることから名付けられたのでしょう。

変数の型

シェルスクリプトには、変数の型はありません。同じ変数に文字列を入れることも、数値を入れることもできます。しかしそのようなプログラミングスタイルはバグを招きやすいため、文字列を入れる変数と数値を入れる変数は分けておくべきです。

また変数に型がないため、変数名はできるだけ内容が理解しやすいものを命名するように心がけてください。var1という名前のシェル変数があったとしても、それはテンポラリの何かの文字列なのか、あるいは一時的なステータス値を保存しているのか、全くわかりません。

一方、user_scoreというシェル変数があれば、それは少なくとも何かのユーザの得点を保持している数値変数なのだろう、とすぐわかります。

パスと最後のスラッシュ

ディレクトリ名を入れる変数の最後に、/(スラッシュ)を入れるか入れないかは、若干好みが分かれるところです。しかし一般的には、スラッシュを入れない書き方が多いようです。この場合、フルパスを組み立てるには自分でスラッシュを挿入する必要があります。

例えば次の例は、シェル変数filepathにはディレクトリ名を入れていますが、最後にスラッシュを入れていないのでファイル名と連結してフルパスを組み立てる際にあいだに/(スラッシュ)を入れています。

```
fullpath="${filepath}/${filename}"
```

このスラッシュの挿入を忘れて、フルパス指定が誤りとなるのはよくある失敗例です。特に同一のスクリプトの中で、最後にスラッシュを入れる変数と入れない変数を混在させるようなことは、誤りを誘発しますから絶対に避けるべきです。

04 cronによるスクリプト実行について

定期的な集計処理やサーバ・ネットワーク監視など、シェルスクリプトを定期的に自動実行したいニーズはさまざまな場面で見られます。UNIXにはそのようなコマンドの自動実行を行うために、cronという仕組みが提供されています。ここで簡単に、cronの動作とその設定方法を説明しておきましょう。

cronの動作と設定ファイル

cronの仕組みは、cronデーモン(Linuxではcrond、FreeBSDではcron)が、設定ファイルcrontabに記載されたとおりにコマンドを定期的に実行する形で提供されます。

システム全体のcronの設定ファイルは、/etc/crontabです。中身はただのテキストファイルなので、読者の環境でもcatコマンドなどで中身を確認してみてください。/etc/crontabは、次のような書式で記述します。このコラムでは、「実行するコマンドの定義」について以下、詳しく説明します。

リスト1 /etc/crontabファイルの設定例

```
# *環境変数の設定
PATH=/sbin:/bin:/usr/sbin:/usr/bin

# *コメントは先頭に#を付ける。
# ---ここはコメント行です。---

# *実行するコマンドの定義
#分 時 日 月 曜日 実行ユーザ    実行コマンド
0   1  *  *  *    user1       /usr/bin/command
```

上記の「実行するコマンドの定義」では、前半の5つのフィールドで、自動実行したい日時設定を指定します。指定可能な日時設定は、次のとおりです。

crontabファイルで指定可能な日時

フィールド	指定値
分	0～59
時	0～23
日	1～31
月	1～12
曜日	0～7 (0と7は日曜日)

各フィールドには、次のような記法が利用できます。

●crontabの各フィールドに可能な記述

記述	意味
*（アスタリスク）	これはそれぞれのカラムにおいて、指定可能な最初の値から最後の値、つまり「すべて」を指す。例えばリスト1は、曜日と月と日が*だから、「毎日 1:00に、user1ユーザ権限で、/usr/bin/command」を実行する
数値の範囲	例えば時フィールドに「8-11」を指定すると、8時、9時、10時、11時に実行する
リスト	例えば時フィールドにカンマで区切って「1,2,5,9」とすると、1時、2時、5時、9時に実行する。リストと範囲は組み合わせが可能
間隔値	「/数値」を入れると、その数値間隔で実行する。例えば分フィールドに「*/5」と指定すると、5分ごとに実行する

ユーザごとのcrontabファイル

/etc/crontabはシステム全体の設定ファイルですから、rootユーザしか編集できません。ユーザごとにcron設定をしたい場合は、**crontabコマンド**に編集を意味する**-eオプション**を付けて「crontab -e」と実行します。こうすることでテキストエディタが起動し、編集後に保存して終了するとユーザごとのcronが設定できます。

「crontab -e」で編集するユーザごとのcron設定は、リスト1の形式から実行ユーザを除いた、次のような形式で記述します。リスト1と比べて、実行ユーザのフィールドがないことに注意してください。

リスト2 ユーザごとのcrontabファイル

```
#分 時 日 月 曜日 実行コマンド

0  1  *  *  *   /usr/bin/command
```

なお、「crontab -e」を実行する際には環境変数EDITORに設定されているエディタが利用されます。

ユーザごとのcrontabファイルは、実際のファイルはLinuxならば「/var/spool/cron/（ユーザ名）」、FreeBSDならば「/var/cron/tabs/（ユーザ名）」となります。記述したユーザごとのcrontabファイルの内容は、crontabコマンドに-lオプションを付けて実行することで確認できます。

●crontab -lによる現在のcrontabの確認

```
$ crontab -l
TMPDIR=/var/tmp

0 1 * * * user1 /usr/bin/command
      :（省略）
```

なお、ユーザごとのcrontabファイルを扱う際の注意点として、crontabコマンドの削除を意味する-rオプションがあげられます。crontabの**-rオプション**は、実行すると警告などを何も表示せず、即座にユーザのcrontabファイルを削除するという動きをします。

crontab -rは警告なしにいきなり削除するため危険

```
$ crontab -r
$ crontab -l
no crontab for user1
```

「crontab -e」でcrontabファイルを変更しようとして、誤って「crontab -r」を実行して丸ごと削除してしまう、というのはありがちな失敗例です。

この失敗の予防策として、「まずcrontab -lで現在の内容をファイルに出力してから、そのファイルを更新して『crontab <ファイル名>』で反映させる。crontab -eは絶対に使わない」という手法をとる人もいます。crontabコマンドは、引数にファイルが指定された場合は、そのファイルの内容を登録するという動作をするためです。

ユーザごとのcron設定の際、望ましいcrontab操作

```
$ crontab -l > ~/crontab
$ vim ~/crontab （図2のように内容を修正する）
$ crontab ~/crontab
```

上記のような手法を利用する際には、修正したファイルをその都度SubversionやGitなどにコミットし、バージョン管理するとさらによいでしょう。

cronのマニュアルについて

少々ややこしいのですが、設定ファイルcrontabと、編集するためのcrontabコマンドは別物です。どちらも同一名称のため、manコマンドでマニュアルを引く際にも混乱しやすいので注意が必要です。具体的なcronのマニュアル類を次にまとめておきましたので参考にしてください。

cron関連のマニュアル類

マニュアルを見るコマンド	マニュアルの対象
man 1 crontab	ユーザーごとのcrontabファイルを管理するcrontabコマンド
man 5 crontab	cron動作のためのcrontab定義ファイルについて
man cron	常駐してcronを動作させるcronデーモンについて

環境変数など

cronから実行されたスクリプトは、コマンドラインから実行した場合とは、環境変数やカレントディレクトリが異なるのが普通です。具体的には、cronから実行した場合には例えばbashならば~/.bashrcが読み込まれません。またカレントディレクトリは、実行ユーザのホームディレクトリとなります。

このような特性を理解していないと、「コマンドライン上にて手で./script.shとして実行すると正常に動作するが、cronに登録して実行するとエラーとなる」という事態に陥りがちです。cronに登録する際は、以下の点に留意してシェルスクリプトを記述するようにしてください。

・カレントディレクトリがどこであっても動作するように書かれているか？
→サンプル023で、これに対応するサンプル例を紹介しています。
・環境変数PATHの値に注意しているか？
→cronでは、例えば/usr/local/binにはパスが通っていないことが普通です。スクリプト実行前にcrontabファイルの中で環境変数PATHに必要なパスを追加するか、あるいはシェルスクリプト内でコマンドをフルパスで書くようにしてください。

Macの場合

現在のMacOSではcronは利用されず、代わりにlaunchdが使われています。そのためMacで定期的にコマンドを実行したい場合には、launchd.plistという設定ファイルを編集します。

launchdの具体的な動作や、launchd.plistファイルの書式などを詳しく説明していると本書の範囲を大きく超えてしまいますので、ここでは省略します。興味のある読者は「man launchd」および「man launchd.plist」として、マニュアルを読んでみてください。

05 pvコマンドのインストール

Linux (CentOS) では、pvコマンドはデフォルトのyumリポジトリには入ってないため、yumコマンドで手軽にインストールすることができません。

またMacOS Xでも、pvコマンドはデフォルトでインストールされていないため、別途インストールする必要があります。

そのためここでは、開発環境を準備してからpvコマンドのソースコードをダウンロードして、システムへインストールするまでの手順を紹介します。

開発環境の準備

ソースコードからインストールするためには、システムの事前準備として、ソースコードを実行可能な形式へと変換(コンパイル)するための開発環境が必要になります。具体的には、**gccコマンド**や**makeコマンド**が必要です。

Linuxではこれらのコマンドは、明示的に入れない設定(インストール時に最小構成を指定するなど)をしない限りは、ほとんどの場合デフォルトでインストールされています。gccコマンドやmakeコマンドに**--versionオプション**を付けてコマンドを実行するとバージョン情報が表示されますので、これでシステムにコマンドがインストールされているか確認してみてください。

gccとmakeコマンドの確認

```
$ gcc --version
gcc (GCC) 4.4.7 20120313 (Red Hat 4.4.7-3)
… (省略)

$ make --version
GNU Make 3.81
… (省略)
```

ここで"command not found"と表示されてエラーになるようなら、これらコマンドをパッケージからインストールしてください。具体的には、Linux (CentOS) ならば**yumコマンド**を利用します。

yumコマンドによるインストール(Linux (CentOS) の場合)

```
# yum install gcc
# yum install make
```

FreeBSDの場合は、デフォルトでgccとmakeはインストールされています。

MacOS Xの場合は、開発環境を利用するためにXcodeというアップルが配布しているデベロッパツールをインストールする必要があります。Xcodeのインストール手順については、バージョンアップごとに大きく手順が変わる可能性があることと、本書の範囲を逸脱してしまうため、ここでは詳しく解説しません。大まかには、まずApple IDを作成し、その後にApp StoreからXcodeをダウンロードしてインストールするという流れになります。

なお、Xcodeのインストール時、Xcodeのバージョンによっては「Command Line Tools」を明示して指定しないとgccやmakeコマンドが入らない場合があるようです。この場合、Xcodeインストール後に、メニューの[Preferences>Downloads]から「Command Line Tools」を追加インストールしてください。

ダウンロード

pvコマンドのソースコードは、以下のWebページからダウンロードすることができます。

pvコマンドのソース

URL http://www.ivarch.com/programs/pv.shtml

このWebページの「Downloads」のところにソースコードへのリンクがあります。ソースコードはファイルの圧縮形式などでいくつか種類がありますが、ここではもっともオーソドックスな、拡張子が.tar.gzになっているファイルをダウンロードしましょう。執筆時点ではバージョン1.4.12が最新版でしたので、ダウンロードしたものはpv-1.4.12.tar.gzです。

ファイルのダウンロードは、Linux/Macの場合はcurlコマンドを使うとよいでしょう。curlコマンドは、システムにデフォルトでインストールされています。

curlコマンドでソースのダウンロード

```
$ curl -O http://www.ivarch.com/programs/sources/pv-1.4.12.tar.gz
```

curlコマンドはデフォルトでは標準出力にダウンロードしたファイルを表示するので、何もオプションを付けないとtar.gzのバイナリが画面にそのまま表示されてしまいます。そのため必ず-O（オー）オプションを付けてファイルとして保存しましょう。

FreeBSDの場合、curlコマンドは標準ではインストールされていません。そのため代わりに、標準でインストールされているfetchコマンドを利用するとよいでしょう。

fetchコマンドでソースのダウンロード

```
% fetch http://www.ivarch.com/programs/sources/pv-1.4.12.tar.gz
pv-1.4.12.tar.gz                              100% of  110 kB   62 kBps
```

インストール

ダウンロードしてきたpv-1.4.12.tar.gzファイルは、gzip圧縮されたtarアーカイブです(このようなソースファイル一式を「tarボール」と呼んだりもします)。そのため、まず**tarコマンド**で展開します。

tarボールの展開

```
$ tar xvzf pv-1.4.12.tar.gz
pv-1.4.12/
pv-1.4.12/doc/
pv-1.4.12/doc/INSTALL
pv-1.4.12/doc/PACKAGE
pv-1.4.12/doc/spec.in
pv-1.4.12/doc/VERSION
…(省略)
```

ここでtarコマンドには、次のオプションを指定しています。これらはよく使いますので覚えておきましょう。これにより、tar.gz圧縮されたアーカイブを展開して、中のファイルを取り出すことができます。

tarコマンドのよく使うオプション

オプション	説明
x	展開
v	展開中のファイルを表示
z	gzipフィルタ
f	ファイルを指定

次にソースコードをコンパイルします。まずソースコードに同梱されている**configureスクリプト**を利用して、Makefileを作成します。続いてmakeコマンドにより実際にソースコードをコンパイルします。最後に、make installで実行ファイルをシステムにインストールします。

こう書くと難しそうですが、実際の手順として必要なコマンドは以下のようにわずかであり、そんなに難しいものではありません。なお、make installコマンドだけは、root権限で実行する必要があります。

pvコマンドのコンパイルとインストール（Linux/FreeBSDの場合）

```
$ cd pv-1.4.12
$ ./configure
$ make
$ su
Password: ―――――――― rootのパスワードを入力
# make install
```

MacOS Xの場合は、最後にsudo make installと実行します。

pvコマンドのコンパイルとインストール（MacOS Xの場合）

```
$ cd pv-1.4.12
$ ./configure
$ make
$ sudo make install
Password: ―――――――― 現在ログインしているユーザのパスワードを入力
```

configureとmakeは一般ユーザで行い、make installのみroot権限で行うのは、誤動作によるシステム破壊のリスクを抑えるためです。これは一種の慣習ですが、ソースコードからインストールする際の決まったお作法として覚えておいたほうがよいでしょう。

インストール先は、デフォルトでは/usr/localとなります。このインストール先を変更したい場合は、configure時に--prefixオプションで変更することができます。

インストール先を指定する例

```
$ ./configure --prefix=$HOME/local
```

上記のようにすると、ホームディレクトリ下にあるlocalディレクトリ内にインストールされます。これならばmake installする際もroot権限は必要ありませんので、まず試しに入れてみる場合は、この手法でホームディレクトリ配下に入れてみるのもよいでしょう。

なおconfigureファイルの中身は、実はシェルスクリプトです。とても大きく複雑なスクリプトなので、読むには骨が折れますが、興味があれば中を見てみてると面白いでしょう。

インストール確認

make installでpvコマンドがインストールできたら、--versionオプションを指定してコマンドを実行してみましょう。

インストールできたことを確認する

```
$ pv --version
pv 1.4.12 - Copyright(C) 2012 Andrew Wood <andrew.wood@ivarch.com>

Web site: http://www.ivarch.com/programs/pv.shtml

This program is free software, and is being distributed under the
terms of the Artistic License 2.0.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

こうしてバージョン情報が表示されれば、正しくインストールできています。
ここで、"command not found"と表示される場合にはインストールできていません。

インストール失敗時

```
$ pv
-bash: pv: command not found
```

なお、インストールの際に-prefixオプションでホームディレクトリ配下のディレクトリを指定してインストールした場合は、おそらくパスが通っていないためにエラーとなるでしょう。この場合は、pvコマンドをフルパスで指定して実行してみてください。

localディレクトリにインストールしたときの例（user1というユーザ名の場合）

```
$ /home/user1/local/bin/pv --version
```

ホームディレクトリ配下にインストールした場合は、シェルスクリプト中でもpvコマンドはフルパスで書く必要があることに注意してください。

06 dialogコマンドのインストール

サンプル004で使用したdialogコマンドは、FreeBSDにはデフォルトでインストールされています。またLinux（CentOSおよびUbuntu）ならばパッケージが用意されているので簡単にインストールできますが、MacOS Xの場合はソースコードをダウンロードして、自分でコンパイルしてシステムへインストールする必要があります。

そのためここでは、Linux（CentOSおよびUbuntu）の場合とMacOS Xの場合のインストール方法についてそれぞれ解説します。

Linux(CentOS)の場合

標準のyumリポジトリに入っていますので、root権限でyum install dialogを実行するだけでインストールできます。

yumコマンドによるインストール

```
# yum install dialog
```

インストールの際、追加で依存するパッケージを入れるかどうか聞かれる場合がありますが、[y]を押せばそれら含めて自動的にインストールしてくれます。

Linux(Ubuntu)の場合

dialogコマンドは公式リポジトリで提供されていますから、**apt-getコマンド**を実行するだけでインストールできます。

apt-getコマンドによるインストール

```
$ sudo apt-get install dialog
```

なおapt-getコマンドでパッケージをインストールするにはroot権限が必要です。Ubuntuではroot権限が必要な作業はsudoコマンドで行いますので、例のように「sudo apt-get」を実行し、パスワードには現在ログイン中のユーザのパスワードを入力してください。

MacOS Xの場合

ソースコードからインストールするには、まず開発環境の準備が必要です。これについ

ては、「pvコマンドのインストール」の「開発環境の準備」を参照してください。

dialogコマンドのソースコードは、以下のWebページからダウンロードすることができます。

dialogコマンドのソース

URL http://invisible-island.net/dialog/dialog.html

このWebページの左メニュー「Download」から、「The source (http)」にソースコードへのリンクがあります。ファイルのダウンロードは、curlコマンドを使うとよいでしょう。curlコマンドは、システムにデフォルトでインストールされています。

curlコマンドでソースのダウンロード

```
$ curl -O http://invisible-island.net/datafiles/release/dialog.tar.gz
```

curlコマンドはデフォルトでは標準出力にダウンロードしたファイルを表示するので、何もオプションを付けないとtar.gzのバイナリが画面にそのまま表示されてしまいます。そのため必ず-Oオプションを付けてファイルとして保存しましょう。

続いて、ダウンロードしたソースコードを、tarコマンドで展開し、configure/make/make installの手順でコンパイルしてインストールします。この具体的な手順については、P.402と同様ですので、そちらを参照してください。

インストール後の確認

dialogコマンドがインストールできたら、--versionオプションを指定してコマンドを実行してみましょう。

インストールできたことを確認する

```
$ dialog --version
Version: 1.2-20121230
```

こうしてバージョン情報が表示されれば、dialogコマンドが正しくインストールできています。一方、コマンドが見つからなければ「command not found」というエラーメッセージが表示されますので、正しくインストールは行えているか、環境変数PATHにインストール先のディレクトリが登録されているかの2点を確認してみてください。

07 setコマンドの利用

setコマンドは多くの機能を持ち、またそれらの機能がお互いにあまり関連性のないこともあり、混乱しやすいコマンドです。そのため、ここで使い方を簡単にまとめておきます。

setコマンドには、大別すると主に以下3つの機能があります。これらについて順番に解説します。

1) シェル変数を表示する
2) シェルのオプションを設定する
3) 位置パラメータ($1、$2など)を操作する

シェル変数を表示する

次のようにsetコマンドを引数なしで実行すると、環境変数を含めて現在定義されている変数一覧が表示されます。

引数なしのsetコマンドは変数一覧を表示する

```
$ set
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:expand_aliases:extquote:force_fignore:hostco
mplete:interactive_comments:login_shell:progcomp:promptvars:sourcepath
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
…(省略)
```

これは、現在のシェル変数の状態を確認するために便利でしょう。

シェルのオプションを設定する

setコマンドでオプションを設定することで、シェルのさまざまな動作モードを指定できます。本書でも、サンプル021で変数未定義ならばエラーとする-uオプションや、サンプル094で終了ステータスが非ゼロならば終了する-eオプションを紹介しました。

この他にもさまざまなオプションを設定可能です。次のページの表にその一部をあげておきました。

setコマンドのオプション（一部）

オプション	意味
-u	未定義の変数を参照するとエラーとする
-e	コマンドが失敗（終了ステータスが非ゼロ）であれば即座にスクリプトを終了する
-n	コマンドを解釈するだけで実行しない。文法チェックに利用する
-f	パス名展開を無効にする。例えば*は展開されず、そのまま*という文字として扱われる
-x	コマンドを展開した結果を表示する。デバッグ時に利用する
-C	リダイレクト時に既存ファイルを上書きしない

また「-o <オプション名>」とすることでもオプションを設定できます。例えば次は、-fと同様のオプション設定を-oで書いたものです。

```
# これは set -f と同じ意味
set -o noglob
```

これらのオプションについて詳しくは、man shとして、組み込みコマンドsetのマニュアルを読んでみてください。

位置パラメータを操作する

通常、位置パラメータ（$1, $2, ...）にはコマンドライン引数が代入されていますが、setコマンドでこの値を設定しなおすことができます。リスト1がsetコマンドで位置パラメータを設定している例です。この場合、setコマンドの引数"Osaka" "Kyobashi",...がそのまま位置パラメータ（$1, $2, ...）となります。そのためリスト1は、コマンドライン引数に何が指定されても、それに関係なく$2である"Kyobashi"を出力します。

リスト1 位置パラメータをsetコマンドで設定

```
#!/bin/sh

set "Osaka" "Kyobashi" "Tamatsukuri"
echo $2
```

なお位置パラメータを操作する際、最初の引数にハイフンで始まる文字列がある場合には、それはオプションとみなされてしまいます。リスト2は、単なる文字列として"-all"を扱いたいケースであると仮定します。

リスト2 指定する引数の最初に、ハイフンで始まる文字列がある場合

```
#!/bin/sh

set "-all" "Osaka" "Kyobashi" "Tamatsukuri"
echo $2
```

しかしリスト2を実行すると、次のように最初の"-all"がオプション扱いされ、エラーとなります。

setコマンドの引数にハイフン付き文字列でエラー

```
$ ./list2.sh
./list2.sh: line 3: set: -l: invalid option
set: usage: set [--abefhkmnptuvxBCHP] [-o option-name] [arg ...]
```

これを防ぐには、リスト3のように--の後ろに引数を指定します。

リスト3 指定する引数の最初に、ハイフンで始まる文字列がある場合

```
#!/bin/sh

set -- "-all" "Osaka" "Kyobashi" "Tamatsukuri"
echo $2
```

リスト3のようにsetコマンドの後ろに--を置くと、それ以降の文字列はオプションとはみなされなくなります。そのため"-all"も単なる文字列として扱われ、オプションとはみなされません。

またこの応用として、リスト4のように「set --」として--だけ指定して引数には何も書かないと、位置パラメータをすべてクリアすることができます。

リスト4 位置パラメータをクリアするので何も表示されない

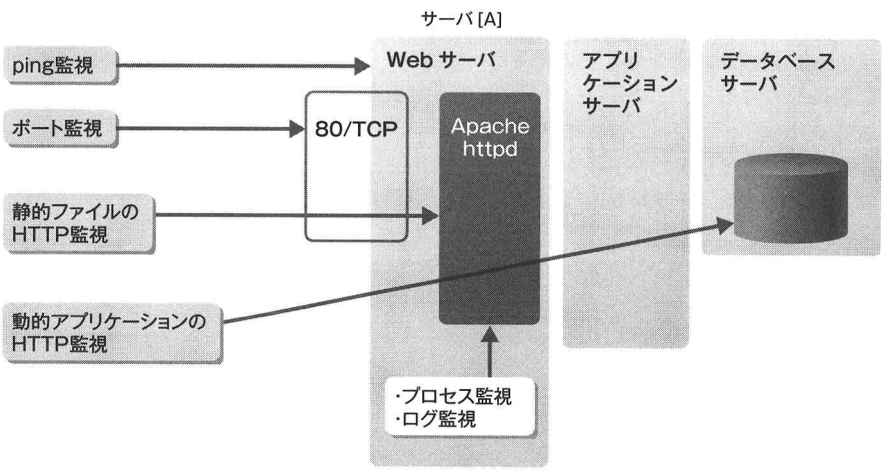
```
#!/bin/sh

set --
echo $2
```

08 Webサービスの監視について

サンプル115でも触れましたが、Webサービスにおいて監視はとても重要な要素です。自分が担当するシステムで、気がつかないうちにいつの間にかサーバが落ちていた、という経験を持つ読者は多いでしょう。サーバ監視をしておけば、そのような状態を素早く発見し対応することができます。しかしひと口に「監視」と言ってもさまざまな手法があり、必要な局面も変わります。

ここでは、次のような構成を例に、シェルスクリプトでの監視手法と適用ケースについて簡単に解説します。

Webアプリケーションの監視



上図に見るように、このWebサービスは静的ファイルをApacheで返し、一部のリクエストはバックエンドのアプリケーションサーバに送ってデータベースから値を取得し、動的にレスポンスを返すものとします。すなわち、サーバ[A]のApacheはWebサーバ兼リバースプロキシサーバとして動作しています。

ここで上記のサーバ[A]について行う監視は、以下のようなものが考えられます。

❖ping監視

Webサーバへpingコマンドを実行し、そのICMPパケットの応答でサーバが起動しているかどうかを監視します。アプリケーションの状態は関係なく、サーバが起動しているかどうかだけを監視するため、死活監視とも呼ばれます。Webサーバのプロセスが落ちて

いても、OSが起動していれば正常と見なされます(→サンプル114)。

❖プロセス監視

サーバ上でpsコマンドを実行し、プロセスが稼働しているかを確認します。ネットワーク的なテストは行わないため、例えばファイアウォールの設定が誤っていて外部からつなげない場合にも、正常とみなされます(→サンプル111)。

❖ログ監視

サーバ上で出力されるログファイルをgrepコマンドなどで解析し、エラーがでていないかをチェックします。クライアント側には通知されない、アプリケーション内部のエラーも見つけることができます。

❖ポート監視

TCPやUDPの特定ポートでサーバまで到達するかを調べるネットワーク監視です。途中経路のネットワーク障害やファイアウォールの誤設定を検知できます。具体的にはncコマンドを利用するとよいでしょう(→サンプル061)。

❖HTTPヘッダ監視

curlコマンドの-Iオプションを用いて、HTTPのHEADメソッドでWebサーバにアクセスできるかを確認します。ファイルのダウンロードは行わずにHTTPヘッダのみ取得するため、サーバ・ネットワークへの負荷が軽くて済みます(→サンプル110)。

❖静的ファイルのHTTP監視

curlコマンドなどでファイルを正常にダウンロードできるかを監視します。実際にファイルをダウンロードするためユーザ環境に近い監視が行えますが、ファイルサイズが大きい場合などには、ネットワークの流量や負荷に注意が必要です(→サンプル115)。

❖動的アプリケーションのHTTP監視

curlコマンドなどでWebアプリケーションにアクセスし、バックエンドサーバから正常にデータを取得して結果を正しくダウンロードできるかどうかまでをチェックします。サービスインした後の運用フェーズでのサービスでは、これが基本的な監視になるでしょう。この場合には、単にアクセスできたかどうかだけではなく、HTTPステータスコードから、成功か否かを判断する必要があります(→サンプル115)。

バックエンドサーバの監視

前ページの図においては、アプリケーションサーバやデータベースサーバも監視が必要です。これらについては、ping監視やプロセス監視など基本的な監視に加えて、CPU・メモリ・ディスク使用量などをチェックするリソース監視が重要でしょう。

例えばアプリケーションサーバはメモリを大量に使うため、スワップが発生していない

かを監視する必要があります(→サンプル117)。常時スワップが発生しているようでは、ディスクI/O待ちとなる時間が長くなり、サーバのロードアベレージ(負荷)が大きくなります。結果、ユーザへのレスポンスが遅くなるでしょう。

データベースサーバについては、データベース領域のディスク使用率が100%となるとデータベース破損など重大な障害を招く危険性があるため、特にディスク使用率の監視は重要です(→サンプル116)。またデータベースは重要なシステムですから、mysqlコマンドを定期的に実行してMySQLサーバプロセスに接続できるかどうかをチェックするDB接続監視を行うこともあります。

運用フェーズでの監視

ポート監視などは、低レイヤでの監視です。例えばサーバ構築時、ファイアウォールの設定を試行錯誤しているときにこの監視がNGになれば、「誤った設定を投入した」などがすぐにわかるでしょう。

一方、サービスイン後には、「動的アプリケーションのHTTP監視」がもっとも高レイヤのため、最低限この監視は行うべきでしょう。システムのどれかの要素が障害を起こし正常にサービスできていないという事態を素早く知ることができます。また、それ以外の監視も、補助的に行うべきです。

いざ障害が発生して「動的アプリケーションのHTTP監視」がNGとなったときには、状況把握のために、より低いレイヤの監視結果がトラブルシューティングに役立ちます。例えば、「プロセス監視は正常だが、ポート監視がNGとなる」という結果がわかれば、おそらく途中経路のネットワークに障害があるか、もしくはファイアウォール設定に誤りがあってTCPポート80をブロックしてしまったのでは、と判断できるでしょう。

なお本書ではシェルスクリプトでの監視例をあげましたが、サーバ監視については、Zabbix、Nagios、Hinemos、Muninなどのオープンソースソフトウェアが提供され広く利用されています。読者がより本格的な大規模サービスを提供するならば、これらのソフトウェアが役立つことでしょう。

Zabbix
URL http://www.zabbix.com/jp/

Nagios
URL http://www.nagios.org/

Hinemos
URL http://www.hinemos.info/

Munin
URL http://munin-monitoring.org/

09 bashのインストールについて

本書の一部では、bash専用の記法を用いた例を紹介しています。MacやLinuxではbashはデフォルトでインストールされていますが、FreeBSDではデフォルトではインストールされません。また、最小構成のインストールオプションを選んだLinux環境などで、bashがない場合があるかもしれません。

システムにbashが存在しない場合には、以下のようにしてbashコマンドのインストールを行ってください。FreeBSDおよびLinux (CentOS) でプロンプトが#となっているところはrootでログインするか、suコマンドを利用して (求められるパスワードを入力して) から、rootユーザで実行します。

bashのインストール：FreeBSDの場合

```
# cd /usr/ports/shells/bash
# make install
```

bashのインストール：Linux (CentOS) の場合

```
# yum install bash
```

Linux (Ubuntu) の場合

```
$ sudo apt-get install bash
```

インストールできたら、bashコマンドを、バージョン番号を確認する--versionオプションを付けて実行してみてください。正常にインストールできていれば、次のようにbashのバージョンが表示されます。

bashコマンドの確認

```
$ bash --version
GNU bash, version 4.1.2(1)-release (x86_64-redhat-linux-gnu)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.
html>

This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

なお実際にbashのシェルスクリプトを書くには、シバン(サンプル070を参照)にbashのパスを記述する必要があります。そのため、bashコマンドがシステムにインストールされているパスを正確に知る必要があります。次のようにwhichコマンドを利用すると、bashコマンドがどこにあるのかを表示することができます。

bashのインストールパスの表示

```
$ which bash
/bin/bash
```

上記の例では、/bin/bashにインストールされていますから、シバンは「#!/bin/bash」と書くことになります。bashは一般的には、LinuxやMacOSでは/bin/bashに、FreeBSDでは/usr/local/bin/bashにインストールされます。

10 参考文献

本書を執筆するにあたり、以下の書籍を参考にさせていただきました。

●「入門UNIXシェルプログラミング シェルの基礎から学ぶUNIXの世界」
Bruce Blinn
SBクリエイティブ
ISBN 978-4797321944

●「UNIXシェルスクリプトハンドブック」
関根達夫
SBクリエイティブ
ISBN 978-4797326529

●「シェルスクリプト基本リファレンス」
山森丈範
技術評論社
ISBN 978-4774146430

●「入門bash 第3版」
Cameron Newham, Bill Rosenblatt
オライリージャパン
ISBN 978-4873112541

●「UNIXシェルスクリプト逆引き大全333の極意」
中橋一朗
秀和システム
ISBN 978-4798008844

●「サーバ/インフラを支える技術」
安井真伸, 横川和哉, ひろせまさあき, 伊藤直也, 田中慎司, 勝見祐己
技術評論社
ISBN 978-4774135663

INDEX

■D～E

■F～G

■H～J

■あ行

■か行

■さ行

■サポートサイト

URL http://isbn.sbcr.jp/77620/

本書をお読みになったご感想、ご意見を上記URLからお寄せください。また、本書のサンプルデータについてもご案内しております。

■注意事項

○本書内の内容の実行については、すべて自己責任のもとで行ってください。内容の実行により発生したいかなる直接、間接的被害について、筆者およびSBクリエイティブ株式会社、製品メーカー、購入した書店、ショップはその責を負いません。
また、本書の内容に関する個別の質問、問い合わせに対し、筆者およびSBクリエイティブ株式会社はその回答の責を追わないものとさせていただきます。

○本書の内容に関するお問い合わせに関して、編集部への電話によるお問い合わせはご遠慮ください。

○お問い合わせに関しては、封書のみでお受けいたします。なお、質問の回答に関しては原則として著者に転送いたしますので、多少のお時間を頂戴、もしくは返答できない場合もありますのであらかじめご了解ください。また、本書を逸脱したご質問に関しては、お答えいたしかねますのでご了承ください。

UNIX(ゆにっくす)シェルスクリプト マスターピース132

2014年6月30日　初版第1刷発行

著　者　大角(おおすみ)　祐介(ゆうすけ)
発行者　小川　淳
発行所　SBクリエイティブ株式会社
〒106-0032 東京都港区六本木2-4-5 興和六本木ビル
TEL 03-5549-1201 (営業)
http://www.sbcr.jp/
印　刷　株式会社 シナノ

装　丁　荒木　慎司
組　版　三門　克二 (株式会社コアスタジオ)

落丁本、乱丁本は小社営業部にてお取替えいたします。
定価はカバーに記載されております。

Printed in Japan　ISBN978-4-7973-7762-0

9784797377620

1920055025003

ISBN978-4-7973-7762-0

C0055 ¥2500E

定価 本体2,500円 +税