誰にでも書ける #! /bin/sh 講座
第 5 回
「立つ鳥跡を濁さず」

安岡孝一

yasuoka : root さん、 root さん。
root : 何だい？
yasuoka : 今日はシェルでテンポラリファイルを残さない方法を、教えてくれるっていう約束だったじゃないですか。
root : ああ、そうだったね。じゃ、まずは前回の users を見せてくれるかい？
yasuoka : はい。

```
% cat users （ぽこ）
#! /bin/sh
# "users" Version 1.0

TEMP1=/tmp/users$$a
TEMP2=/tmp/users$$b
who > $TEMP1
cp /dev/null $TEMP2
set ‘wc $TEMP1‘
LINE=$1

while [ $LINE -gt 0 ]
do set ‘head -$LINE $TEMP1 | tail -1‘
   echo $1 >> $TEMP2
   LINE=‘expr $LINE - 1‘
done

sort $TEMP2 | uniq | pr -t -8 -w80 -l1
rm $TEMP1 $TEMP2
exit 0
% ■
```

root : つまり

```
% users （ぽこ）
（コントロール C）
% ■
```

で users を無理矢理止めちゃうと

```
% pushd /tmp （ぽこ）
/tmp ~/bin
% ls （ぽこ）
users1422a      users1422b
% cat users1422a （ぽこ）
yasuoka  console Mar  4 16:02
hama     ttyp1   Mar  4 15:09   (daikaku:0.0)
takahash ttyp3   Mar  3 16:20   (kinkaku:0.0)
% cat users1422b （ぽこ）
takahash
% ■
```

っていうテンポラリファイルが /tmp に残っちゃうのを、何とかしようと。

```
% rm users* （ぽこ）
rm: remove users1422a? y （ぽこ）
rm: remove users1422b? y （ぽこ）
% popd （ぽこ）
~/bin
% ■
```

yasuoka : ええ。
root : じゃあ、 users をこんな風に書き換えてごらん。

```
#! /bin/sh
# "users" Version 1.1

TEMP1=/tmp/users$$a
TEMP2=/tmp/users$$b
trap "rm -f $TEMP1 $TEMP2 ; exit 1" 2
```

```
who > $TEMP1
cp /dev/null $TEMP2
set ‘wc $TEMP1‘
LINE=$1

while [ $LINE -gt 0 ]
do set ‘head -$LINE $TEMP1 | tail -1‘
   echo $1 >> $TEMP2
   LINE=‘expr $LINE - 1‘
done

sort $TEMP2 | uniq | pr -t -8 -w80 -l1
rm $TEMP1 $TEMP2
exit 0
```

（間）

yasuoka ： できました。よく見ると trapってのが 6 行目に加わっただけですね。

root ： うん、そうだよ。でもこれで、コントロール C を実行中に押されても、ちゃんとテンポラリファイルを rm する。

```
% users（ぽこ）
（コントロール C）
% ls /tmp（ぽこ）
% 
```

yasuoka ： あ、ほんとだ。どういう仕掛けなんですか？

```
trap 文字列 シグナル番号
      シグナルを受けとった際に実行するコマンド列を、文字列として設定する。
      シグナル番号は複数書いてもよい。
一般的なシグナル番号の意味は以下の通り。
        1    端末からのブレーク信号
        2    キーボードからの割り込み
        3    キーボードからの強制終了
        8    演算例外
        9    非常強制終了（trap できない）
       10    バスエラー
       11    セグメンテーションフォールト
       13    パイプの行き先がない
       15    一般強制終了
```

root ： キーボード割り込みってのは、プロセスにシグナルの 2 番を送るんだ。そこでシグナルの 2 番に対して trap を書いておくと、キーボード割り込みが起こった時にそこに書かれているコマンドを実行する。

yasuoka ： それが rm -f $TEMP1 $TEMP2 ; exit 1 ですね。でもこの;って何ですか？

root ： コマンドの区切り記号だよ。1 行に複数のコマンドを書く時には、; で区切るんだ。ついでだから、他の区切り記号も教えとこう。

```
コマンド ; コマンド
      前のコマンドに続いて後のコマンドを実行。単なる改行はこれに同じ。
コマンド & コマンド
      前のコマンドの終了を待たずに、後のコマンドを実行。$! に前のコマンドのプロセス番号が入る。
コマンド | コマンド          あるいは          コマンド ^ コマンド
      前のコマンドの標準出力を、後のコマンドの標準入力に繋ぐ。
コマンド && コマンド
      前のコマンドのエクジットステイタスが 0 だったら、後のコマンドを実行。
コマンド || コマンド
      前のコマンドのエクジットステイタスが 0 以外だったら、後のコマンドを実行。
```

yasuoka ： へーえ。でもどうして exit 1 が要るんですか？

root ： trap では文字列のところを実行した後、実行中だったところへ戻ってしまうからね。この exit を書いておかないと、 rm した後もプログラムを続行してしまう。

yasuoka ： そうですか。ところで、キーボード割り込み以外にもシグナルを送る方法はあるんですか？

root ： キーボードからの強制終了とかもあるけど、やっぱり kill だね。

```
kill -シグナル番号 プロセス番号
      プロセスにシグナルを送る。プロセス番号は複数書いてもよい。「-シグナル番号」が省略された場合は、 15 番のシグナルが送られる。
```

root : だから 6 行目では、2 番だけでなく 3 番や 15 番も trap しておいたほうがいい。

yasuoka : そうですか。だんだん複雑になってきますね。

root : そうだね。trap を使うってのは、ひとつ間違うと止まらないプログラムになっちゃうからね。

yasuoka : でも、trap を使わずにテンポラリファイルを消す方法ってありますか？

root : この users ならなんとかなるね。

yasuoka : え？

root : 始めから、テンポラリファイルを使わないプログラムにすればいいんだ。

```
#! /bin/sh
# "users" Version 2.0

who |
( while read LINE
  do set $LINE
     echo $1
  done
) | sort | uniq | pr -t -8 -w80 -l1

exit 0
```

yasuoka : こんなに短くなるんですか？ しかもテンポラリファイルも使わずに。

root : うん。やっぱりサブシェルの効果は絶大だね。

(コマンド列)
サブシェルを立ちあげる。括弧で囲まれるコマンド列全体を 1 つのコマンドとみなして、標準入力、標準出力等を繋ぐことができる。なお括弧内での cd や変数への代入は、括弧外に影響しない。

yasuoka : いくつか質問していいですか？

root : ああ、いいよ。

yasuoka : まず、4 行目の who の後のパイプが、どこにも繋がってないように見えるんですけど。

root : ああ、それは次の行の (while read LINE に繋がってるんだよ。区切り記号の後の改行は無視されるからね。

yasuoka : それから while read LINE のループが、いつ終了するのかわかりません。

root : read のエクジットステイタスは、いつ 1 になる？

yasuoka : エンドオブファイルです。

root : 正解。だから who から来た入力が終った時点で終了だ。それまでは、LINE に 1 行づつ代入されて処理される。

yasuoka : そして、ユーザ ID の部分だけが sort 以降に渡されるんですね。うーん、すごい。

```
% users （ぽこ）
hama      takahash  yasuoka
% 
```

ところで root さん、もう一つ見てほしいプログラムがあるんですけどいいですか？

root : ああ、いいよ。

yasuoka : ちょっと長いんですけど。

```
% cat dow （ぽこ）
#! /bin/sh
# "dow" Version 1.0

if [ $# -ne 3 ]
then echo Usage: dow month day year >&2
     exit 1
fi

DAY=‘expr $2 + $3 + $3 / 4 - $3 / 100 + $3 / 400‘
if expr $3 % 4 > /dev/null
then LEAP=false
elif expr $3 % 100 > /dev/null
then LEAP=true
elif expr $3 % 400 > /dev/null
then LEAP=false
else LEAP=true
fi
```

```
fgrep :$1 << 'EOF' |
6 5 :January:1
2 1 :February:2
2 2 :March:3:November:11
5 5 :April:4:July:7
0 0 :May:5
3 3 :June:6
1 1 :August:8
4 4 :September:9:December:12
6 6 :October:10
EOF
( if read LINE
  then set $LINE
  else echo Usage: dow month day year >&2
       exit 1
  fi
  if $LEAP
  then DAY=`expr '(' $2 + $DAY ')' % 7 + 1`
  else DAY=`expr '(' $1 + $DAY ')' % 7 + 1`
  fi
  set Sun Mon Tues Wednes Thurs Fri Satur
  eval echo '$'$DAY'day'
  exit 0
)
exit
% ■
```

root : ふーん、曜日を計算するプログラムか。やってみせてくれるかい？

yasuoka : はい。

```
% dow Mar 4 1990 （ぽこ）
Sunday
% ■
```

こんな感じです。

root : で、どこがわからないの？

yasuoka : 19 行目の `fgrep :$1 << 'EOF' |`ってのがよくわかりません。特にパイプがどこに繋がってるのかが。

root : `$1` には月の名前が入ってるよね。

yasuoka : ええ。

root : で、 20 行目から 28 行目までを `fgrep` の標準入力に繋いでるだろ。

yasuoka : え？

root : 忘れたのかい？ 第 2 回のリダイレクトのところを、もう一度よく読んでおくように。

yasuoka : はい。

root : で、指定された月を含む行を取り出して、それを 30 行目からのサブシェルの標準入力に繋いでるんだ。

yasuoka : あ、そういうことですか。それでうまく取り出せなかった時には、 `if read LINE` が `else` に行くから、 `exit 1` と。この `if $LEAP`って何ですか？

root : `LEAP`って変数には何が入ってる？

yasuoka : 10 行目から 17 行目の間で、 `true` か `false` を入れてるみたいなんですけど。

root : そう、閏年なら `true`、そうでなければ `false` が入ってるね。実は `true` も `false` もコマンドなんだ。

`true`
　　エグジットステイタスとして 0 を返す。出力なし。
`false`
　　エグジットステイタスとして 0 以外を返す。出力なし。

root : こうすると `LEAP` が `true` なら `then` 以降が、 `false` なら `else` 以降が実行されるからね。これを使って、閏年の 1 月と 2 月を修正してるんだな。

yasuoka : ははーん。で、最後から 4 行目なんですけど。

root : ああ、 `eval` は教えてなかったね。

`eval` 文字列
　　文字列を評価した後、実行する。複数の文字列を書いてもよい。

root : `DAY`っていう変数の値が例えば 2 だったら、最後から 4 行目は

```
echo $2day
```

ていう風に評価されてから実行される。 1 つ前の行で `$2` には `Mon` が入ってるから、結局この例では `Monday` が出力される。

yasuoka : 変数の間接参照ができるんですね。

root ： まあ、そうかな。

```
% dow Sep 1 1752 （ぽこ）
Friday
% 
```

でもこの dowってプログラム、 1752 年 9 月 14 日を考慮に入れてないね。

yasuoka ： 何ですか、その 1752 年 9 月 14 日って？

root ： カレンダーを出力するコマンドは知ってるかい？

yasuoka ： いいえ。

root ： calっていうんだけどね。

```
% cal 3 1990 （ぽこ）
    March 1990
 S  M Tu  W Th  F  S
                1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31

% 
```

cal 月 年 カレンダーを出力する。出力は標準出力。

root ： これで、 cal 9 1752 としてごらん。

yasuoka ： 1752 年 9 月のカレンダーですか。

```
% cal 9 1752 （ぽこ）
   September 1752
 S  M Tu  W Th  F  S
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30


% 
```

あれ？ 何ですか、これ？

root ： イギリスでは 1752 年 9 月 14 日にグレゴリオ暦を採用したからね。その年の 9 月は 3 日から 13 日がなかったんだ。 cal はそれに合わせてるんだよ。

yasuoka ： それ以前は太陰暦だったんですか？

root ： いや、ユリウス暦だったんだよ。「400 年に 3 回、閏年を抜く」ってのをやらないやつ。

yasuoka ： へーえ。でもそれなら日本では 1873 年の改暦をやらないと。

root ： まあね。ただそれ以前の宝暦暦とか天保暦とかを処理するのは、ちょっとむずかしいだろうな。曜日っていう概念があったのかどうかも怪しいし。確か暦のおおもとのローマじゃ、ユリウス / グレゴリオ改暦は 1582 年 10 月 15 日で、 10 月 5 日から 14 日がなかったはずだ。

yasuoka ： だったらむしろローマの方に合わせたいですね。

root ： ふむ。

yasuoka ： で、 root さん。

root ： 何だい？

yasuoka ： さっきの dow、年を省略した場合に今年をデフォルトにできませんか？

root ： date を使えば簡単だろう。

date 現在の日付、時刻を出力する。出力は標準出力。

yasuoka ： あ、そうですね。

```
% date （ぽこ）
Sun Mar  4 18:23:01 JST 1990
% 
```

root ： それさえわかれば、もうあとは自分で書けるだろう？

yasuoka ： ええ。何とかなると思います。

root ： よし。これからは頑張って、いいシェル・スクリプトを書くんだよ。

yasuoka ： はい。どうもありがとうございました。