続・誰にでも使える csh 講座
第 3 回
「path と PATH」

安岡孝一

yasuoka ： root さん、 root さん。

root ： 何だい？

yasuoka ： 前から思ってたんですけど、 C シェルでコマンドパスを表す変数には、 path と PATH と 2 つありますよね。

```
~% echo $path （ぽこ）
/home/yasuoka/bin /usr/local/bin /usr/ucb /bin /usr/bin .
~% echo $PATH （ぽこ）
/home/yasuoka/bin:/usr/local/bin:/usr/ucb:/bin:/usr/bin:.
~% ■
```

で、 path を変更すると PATH にも反映されるけど

```
~% set path=(. ~/bin /usr/ucb /bin /usr/bin) （ぽこ）
~% echo $path （ぽこ）
. /home/yasuoka/bin /usr/ucb /bin /usr/bin
~% echo $PATH （ぽこ）
.:/home/yasuoka/bin:/usr/ucb:/bin:/usr/bin
~% ■
```

PATH を変更しても path には反映されない。

```
~% set PATH=/usr/ucb:/bin:/usr/bin （ぽこ）
~% echo $PATH （ぽこ）
/usr/ucb:/bin:/usr/bin
~% echo $path （ぽこ）
. /home/yasuoka/bin /usr/ucb /bin /usr/bin
~% ■
```

これってどうなってるんですか？

root ： PATH は環境変数だよ。 set で代入するのは変だ。

yasuoka ： え？

root ： unset PATH して echo $PATH してごらん。

yasuoka ： unset PATH ですか？

```
~% unset PATH （ぽこ）
~% echo $PATH （ぽこ）
.:/home/yasuoka/bin:/usr/ucb:/bin:/usr/bin
~% ■
```

あれ、 unset したはずなのに戻ってる。

root ： そこで setenv PATH /usr/ucb:/bin:/usr/bin して、 PATH と path を見てごらん。

yasuoka ： setenv ですか？

```
~% setenv PATH /usr/ucb:/bin:/usr/bin （ぽこ）
~% echo $PATH （ぽこ）
/usr/ucb:/bin:/usr/bin
~% echo $path （ぽこ）
/usr/ucb /bin /usr/bin
~% ■
```

あ、反映された。どういうことなんですか？

setenv 環境変数 文字列 　　C シェル内蔵。環境変数に文字列を代入する。 setenv 　　C シェル内蔵。全ての環境変数の情報を標準出力に出力する。 unsetenv 環境変数 　　C シェル内蔵。環境変数をセットされていない状態に戻す。

root ： C シェルでコマンドパスを表すのは、確かに path と PATH の 2 種類があるけど、 path は普通の変数で、 PATH は環境変数なんだ。

yasuoka ： 環境変数って何ですか？

root ： コマンドとかスクリプトの中に持ち込まれる変数。

yasuoka ： 持ち込まれるって？

root ： うーん、実際に見てみた方が早いな。シェルを立ち上げてごらん。

yasuoka ： はい。

```
~% sh （ぽこ）
$ ■
```

root : PATH と path はどうなってる？

yasuoka : えっと

```
$ echo $PATH （ぽこ）
/usr/ucb:/bin:/usr/bin
$ echo $path （ぽこ）

$ 
```

PATH は設定されてるけど、 path は設定されてません。

root : で、 PATH の内容が、さっき setenv したのと同じになってるだろ？ それが環境変数。

yasuoka : C シェルの setenv で代入した値が、そこから実行するコマンドとかに引き継がれるんですか？

root : そう。ちょっと試してごらん。

yasuoka : はい。

```
$ exec true （ぽこ）
~% echo $ok （ぽこ）
ok: Undefined variable.
~% setenv ok 'Tameshini chotto' （ぽこ）
~% echo $ok （ぽこ）
Tameshini chotto
~% sh （ぽこ）
$ echo $ok （ぽこ）
Tameshini chotto
$ 
```

うーん、すごい。

```
$ exec true （ぽこ）
~% unsetenv ok （ぽこ）
~% set ok=Doukana （ぽこ）
~% echo $ok （ぽこ）
Doukana
~% sh （ぽこ）
$ echo $ok （ぽこ）

$ 
```

setenv で代入した環境変数はシェルに引き継がれるけど、 set で代入した変数は引き継がれないんですね。

root : そう。

yasuoka : じゃ、最初の質問に戻るんですけど、どうして C シェルでは、コマンドパスの設定に set path と setenv PATH の 2 つがあるんですか？

root : 元々コマンドパスを表すのは、環境変数の PATH なんだよ。少なくとも C シェルが現れる以前は、そうだったんだ。

yasuoka : はい。

root : ところが C シェルで配列変数が扱えるようになった時に、コマンドパスも配列で表した方が便利なんじゃないか、っていう話になった。パスのディレクトリの順番を変更する、とかいうのが、簡単にできるからね。

yasuoka : そうなんですか？

root : うん。それで pathって配列でコマンドパスを表すことにしたんだけど、他のコマンドとかは相変わらず PATH でコマンドパスを表してるから、 set path を実行した時に同時に PATH も変更しないとまずい、ってことになったんだよ。これが set path と setenv PATH の関係として、今も残ってるわけだ。

yasuoka : ふーん。

```
$ setenv ok Doukana （ぽこ）
setenv: not found
$ 
```

シェルでは環境変数はどうやって設定するんですか？

root : 普通に代入して export だよ。

export 変数 　　シェル内蔵。変数を環境変数とみなす。

yasuoka : こうですか？

```
$ ok=Doukana （ぽこ）
$ export ok （ぽこ）
$ 
```

root : うん、そう。

yasuoka : 環境変数の一覧を見るには？

root : BSD なら printenv だ。

printenv
　　BSD のみ。全ての環境変数の情報を標準出力に出力する。
env
　　System V のみ。全ての環境変数の情報を標準出力に出力する。

yasuoka ： printenv ですね。

```
$ printenv （ぽこ）
HOME=/home/yasuoka
PATH=/usr/ucb:/bin:/usr/bin
SHELL=/bin/csh
TERM=mooncons
USER=yasuoka
ok=Doukana
$ ■
```

変更は効くのかな？

```
$ ok='Tameshini chotto' （ぽこ）
$ printenv | egrep '^ok' （ぽこ）
ok=Tameshini chotto
$ ■
```

あ、 export しなおさなくてもいいんですね。

root ： そうだけど…。 egrep なんて教えたっけ？

egrep 正規表現 ファイル名
　　正規表現にマッチする行を全て出力する。入力はファイル、出力は標準出力。ただしファイル名が省略された場合は、入力は標準入力。
egrep -v 正規表現 ファイル名
　　正規表現にマッチしない行を全て出力する。他は上に同じ。
エグジットステイタスは、出力行があった時には 0、なかった時には 1 となる。使用できる正規表現は、以下の通り。

^	行の先頭
$	行の末尾
.	任意の 1 文字
[複数の文字]	複数の文字のいずれか 1 文字、 - は省略を意味
[^複数の文字]	複数の文字以外の 1 文字、上と同じ省略が可能
*	直前の正規表現の 0 回以上の繰り返し
+	直前の正規表現の 1 回以上の繰り返し
?	直前の正規表現の 1 回以下の繰り返し
正規表現|正規表現	いずれかの正規表現
(正規表現)	正規表現それ自身、 * や | のおよぶ範囲を限定
\^	^
\$	$
\.	.
\[	[
\]	]
*	*
\+	+
\?	?
\\|	|
\(	(
\)	)
\\	\
その他の文字	その文字自身

yasuoka ： でも「ok から始まる行」なんて fgrep じゃ書けないでしょう？
root ： うん。
yasuoka ： それに egrep の正規表現は awk と同じですから。

```
$ exec true （ぽこ）
~% ■
```

それより root さん。シェルを終了する時、いつも exec trueってしてるんですけど、これどういう意味なんですか？

root ： true を実行してシェルを終了する、っていう意味だよ。正確にはちょっと違うけど。

exec コマンド
　　シェルおよび C シェル内蔵。新たなプロセスを生成せず、現在のシェルあるいは C シェルのプロセスがそのままコマンドを実行する。

root ： もう一度シェルを立ち上げてごらん。

yasuoka : はい。

```
~% sh（ぽこ）
$ 
```

root : そのシェル、プロセス番号は何番だい？

yasuoka : えっと

```
$ echo $$（ぽこ）
2008
$ 
```

2008 です。

root : じゃ、 ps してごらん。

yasuoka : ps ですか？

```
$ ps（ぽこ）
  PID TT STAT  TIME COMMAND
 2008 co S     0:00 sh
 2014 co R     0:00 ps
$ 
```

root : 次に exec ps してごらん。

yasuoka : exec ps ですね。

```
$ exec ps（ぽこ）
  PID TT STAT  TIME COMMAND
 2008 co R     0:00 ps
~% 
```

root : 今、 exec ps した時の ps のプロセス番号、何番だった？

yasuoka : 2008 です。あれ？ これって sh のプロセス番号だったはずじゃ…。

root : そう。 exec は、シェルのプロセス自身がそのコマンドに化けて、コマンドを実行するんだよ。だからプロセス番号は、シェルのプロセス番号がそのまま引き継がれる。

yasuoka : ふーん。でもどうしてシェルを終了するのに、わざわざ exec なんて使うんですか？ exit でもよさそうなものなのに。

root : ためしにシェルで exit してごらん。

yasuoka : え？

```
~% sh（ぽこ）
$ exit（ぽこ）
$ 
```

あら、終了しない。

root : ginkaku の sh は、 BSD のシェルだからね。対話的に動いてる時は exit では終了しない。 System V のシェルだと exit で終了するものもあるけど、 exec true なら、どのシェルでも確実に終了できるからね。

yasuoka : そういうことだったんですか。でも、どうして exit で終了しないようになってるんでしょう？

root : . で読んだスクリプトに exit があっても、終了しないようにかなぁ。

source ファイル名 C シェル内蔵。ファイル中のコマンドを実行する。ファイルは C シェルのスクリプトでなければならない。 . コマンド名 シェル内蔵。シェル・スクリプトとして書かれたコマンドを、シェルが直接実行する。

yasuoka : . って何ですか？

root : source のシェル版みたいなものだけどね。~/bin の中にシェル・スクリプトのコマンドはあるかい？

yasuoka : えっと

```
$ cat ~/bin/ascii（ぽこ）
~/bin/ascii: No such file or directory
$ 
```

あれ？

root : ~ がホームディレクトリを意味するのは、 C シェルだけだよ。シェルでは $HOME でも使うしかないな。

yasuoka : そうでしたね。

```
$ cat $HOME/bin/ascii（ぽこ）
#! /bin/sh
# "ascii" Version 1.3

case "$1" in
?) set x "$1" `echo "$1" | od -b`
   echo $4 "$2" ;;
```

```
000) csh -fc "glob '000 ' ''"
     echo '' ;;
[0-3][0-7][0-7]) echo $1 x | tr x '\'$1 ;;
esac

exit 0
$ ■
```

これなんかどうですか？

root : うーん、パラメータが要るなぁ。 `set x` して、`. ascii` してごらん。

yasuoka : `set x ; . ascii` ですね。

```
$ set x ; . ascii （ぽこ）
170 x
$ ■
```

root : ね、スクリプトの最後に `exit 0` があるけど、それでシェルそのものを終了したりはしなかっただろ。で、 `echo $*` してみて。

yasuoka : 全パラメータの出力ですね。

```
$ echo $* （ぽこ）
x x 0000000 170 012 0000002
$ ■
```

変わってる。

root : . のシェル・スクリプトの実行は、 C シェルの `source` と同じで、スクリプトをキーボードから打ち込んだのと同じような動作になる。 `source` と違うのは、ファイルを見つけるのに `PATH` を調べるってことだ。

yasuoka : そうなんですか。

root : でもこれが、対話型シェルで `exit` が効かない本当の理由かどうかはわからない。

yasuoka : 結局、謎なんですね。

```
$ exec whoami （ぽこ）
yasuoka
~% ■
```

そういえば、 System V での `awk` のスクリプトは 1 行目に

```
exec awk "‘sed 1d $0‘" ${1+"$@"}
```

って書くんでしたよね。あれも関係があるんですか？

root : うん。あれは最初はシェル・スクリプトとして起動されるんだよ。で、 `$0` にはコマンドの絶対パスが入ってるから、 `sed 1d` で最初の行が削られて、`exec awk` するわけだ。

yasuoka : じゃあ、 `#! /bin/awk -f` ってのは？

root : こっちは直接

```
/bin/awk -f コマンドの絶対パス パラメータ
```

っていう形で起動される。 BSD では `#!` にそういう意味があるからね。

`awk -f` ファイル名 ファイル名 　　前のファイルを `awk` のスクリプトとして実行する。 `sed -f` ファイル名 ファイル名 　　前のファイルを `sed` のスクリプトとして実行する。 いずれも、入力は後のファイル、出力は標準出力。ただし後のファイル名が省略された場合は、入力は標準入力。

yasuoka : root さん、もう 1 つだけ訊いていいですか？

root : 何だい？

yasuoka : `rehash` って何ですか？

`rehash`
　　C シェル内蔵。コマンドのパス一覧表を現在のものに書き換える。
`hash -r`
　　シェル内蔵、 System V のみ。コマンドのパス一覧表を全て白紙に戻す。

root : C シェルとか System V のシェルとかは、各コマンドがコマンドパスのどこにあるかを記憶しておくハッシュ表っていうのを持ってるんだ。これを使ってコマンドの実行を速くしてるんだけど、でもそういうものを持ってると、新しいコマンドを追加した時には表を書き換える必要がある。そのためのコマンドが C シェルでは `rehash` なんだ。

yasuoka : そういうことですか。それで理解できました。

root : ハッシュ表は `rehash` 以外でも、 `set path` とか `setenv PATH` した時には書き換えられるけどね。さて、色々話してきたけど、コマンドパスに関わる話は大体わかったかい？

yasuoka : はい、よくわかりました。どうもありがとうございました。