シェルを使おう
- 応用編 -

lilo-bk
友國哲男
(tomokuni@netfort.gr.jp)

1.1 仕組み (1)

リダイレクトとパイプの原理及び注意点は次の通りである。
- リダイレクトより先にパイプのファイルディスクリプタ処理を行う。
- 左から順に評価される。
- dup2(2) はオブジェクトの複製をする。
- パイプはサブシェルで実行される。
  - 実装によってはパイプの最後段のみカレントシェルで実行されるものがある。
  - この場合(当然だが)最後段での変数操作の結果がその後も有効である。

1.1 仕組み (2)

(例 1)
$ command >file
- 標準出力が file に出力される。

$ command 2>file
- 標準エラー出力が file に出力される。

$ command >file 2>&1
- 標準出力と標準エラー出力ともに file に出力される。

$ command 2>&1 >file
- 標準出力が file に出力され、標準エラー出力は標準出力(この場合は端末)に出力される。

1.1 仕組み (3)

(例 2)
$ command1 | command2
- command1 の標準出力が command2 の標準出力にパイプで結合される。

$ command1 2>&1 | command2
- まず command1 の標準出力が command2 の標準出力にパイプで結合され、その後パイプの前段の標準出力に command1 の標準エラー出力が結合される。
- 結果として command1 の標準出力と標準エラー出力が command2 の標準出力に結合される。

1.2 標準エラー出力のみをパイプ (1)

パイプの後段に標準エラー出力「のみ」渡すにはどうしたらよいだろうか？

(例 1)
$ command1 2>&1 >/dev/null | command2
- (1) 標準出力がパイプで command2 に接続
- (2) 標準エラー出力を標準出力にリダイレクト
- (3) (もともとの)標準出力を /dev/null にリダイレクト

結果的に
- もともとの標準出力
  - /dev/null へ
- もともとの標準エラー出力
  - command2 へパイプ

となる。

1.2 標準エラー出力のみをパイプ (2)

(例 2)
$ command1 3>&1 >/dev/null 2>&3 | command2
- (1) 標準出力がパイプで command2 に接続
- (2) ダミー(ファイルディスクリプタ 3 番)を標準出力にリダイレクト
- (3) (もともとの)標準出力を /dev/null にリダイレクト
- (4) 標準エラー出力をダミーに→その結果標準出力にリダイレクト

結果的に
- もともとの標準出力
  - /dev/null へ
- もともとの標準エラー出力
  - command2 へパイプ
- ダミー(fd 3)
  - オープンしたまま(コマンド終了後クローズ)

となる。

1.2 標準エラー出力のみをパイプ (3)

(例 3)
$ command1 3>&1 1>&2 2>&3 3>&- | command2
- (1) 標準出力がパイプで command2 に接続
- (2) ダミー(ファイルディスクリプタ 3 番)を標準出力にリダイレクト
- (3) (もともとの)標準出力を標準エラー出力にリダイレクト
- (4) (もともとの)標準エラー出力をダミーに→その結果標準出力にリダイレクト
- (5) ダミー(ファイルディスクリプタ 3 番)をクローズ

結果的に
- もともとの標準出力
  - 端末(画面)へ
- もともとの標準エラー出力
  - command2 へパイプ
- ダミー(fd 3)
  - 一時的にオープンされて処理後クローズ

となる。

1.2 標準エラー出力のみをパイプ (4)

(参考) [lilo:22219] より
ping の結果を more で確認しつつ ping の終了コードを得る場合は、次のようにするとよい。

```
$ stat='exec 4>&1;
> { ping -c hoge 2>&1 4>&-; echo $? 1>&4; } | more 1>&2 4>&-'
$ echo $stat
```

--- ループ文での注意 (1) ---

for や while 等のループ文にもリダイレクトやパイプが使える。
(例)
```
$ for i in *.txt; do echo $i; done > textfile.list
```

ただし、シェルスクリプト中のループ文でリダイレクトやパイプを
使用した場合、 Bourne Shell ではそのループ文がサブシェルで実行
されてしまうので、変数等の扱いには十分注意しなければならない。

(例 1) リダイレクト、パイプ無し
```
$ cat roop1.sh
n=0
while read line
do
  n=‘expr $n + 1‘
  echo "$n: $line"
done
echo "total line = $n"
```


--- ループ文での注意 (2) ---

```
$ ./roop1.sh < roop1.sh
1: n=0
2: while read line
3: do
4: n=‘expr $n + 1‘
5: echo "$n: $line"
6: done
7: echo "total line = $n"
total line = 7

$ cat roop1.sh | ./roop1.sh
1: n=0
2: while read line
3: do
4: n=‘expr $n + 1‘
5: echo "$n: $line"
6: done
7: echo "total line = $n"
total line = 7
```


--- ループ文での注意 (3) ---

(例 2) リダイレクト使用
```
$ cat roop2.sh
n=0
while read line
do
  n=‘expr $n + 1‘
  echo "$n: $line"
done < $0
echo "total line = $n"

$ ./roop2.sh
1: n=0
2: while read line
3: do
4:   n=‘expr $n + 1‘
5:   echo "$n: $line"
6: done < $0
7: echo "total line = $n"
total line = 0
```


--- ループ文での注意 (4) ---

(例 3) パイプ使用
```
$ cat roop3.sh
n=0
cat $0 | while read line
do
  n=‘expr $n + 1‘
  echo "$n: $line"
done
echo "total line = $n"

$ ./roop3.sh
1: n=0
2: cat $0 | while read line
3: do
4:   n=‘expr $n + 1‘
5:   echo "$n: $line"
6: done
7: echo "total line = $n"
total line = 0
```


--- ループ文での注意 (5) ---

roop2.sh や roop3.sh のような場合にも $n を保存する方法はある。
それは exec を使ってファイルディスクリプタを切り替えることで
実現可能である。(次節参照)

(注意)
最近の Bourne Shell 系の Shell (bash,zsh,ash) は
更にこれらとはちがう挙動を示すので、これまた要注意である。

- bash,sh(bash へのリンク),ash,pdksh
  - roop2.sh のみ $n が 7 になる
- zsh
  - roop2.sh, roop3.sh 共に $n が 7 になる

--- 連想配列を使おう (1) ---

bash や zsh 等には既に配列の機能が備わっているが、
(オリジナルの) Bourne Shell ではその機能はない。
しかし eval を使うことで連想配列(まがい?)が実現できる。

(例)
```
$ i=1
$ eval M_$i='"January"'
$ echo $M_1
January
```

これを応用すると例えば次のようなことが可能となる。

--- 連想配列を使おう (2) ---

```
$ cat array.sh
exec 3<&0 0<<EOF
5 Thursday
2 Monday
4 Wednesday
7 Saturday
3 Tuesday
1 Sunday
6 Friday
EOF
i=0
while read j week
do
  i=‘expr $i + 1‘
  eval M_$j='"$week"'
done
exec 0<&3 3<&-
j=1
while test $j -le $i
do
  eval echo \$j -- \$M_$j
  j=‘expr $j + 1‘
done
```


--- 連想配列を使おう (3) ---

実行すると、、、

```
$ ./array.sh
1 -- Sunday
2 -- Monday
3 -- Tuesday
4 -- Wednesday
5 -- Thursday
6 -- Friday
7 -- Saturday
```

とヒアドキュメントの部分がソートされて出力される。

3 eval を用いた連想配列

--- 連想配列を使おう (4) ---

(注意)
前節で少し触れたが、このスクリプト中の連想配列に格納している
while 文に対して、パイプやリダイレクトを用いると後方で変数参照
できないために、 1 行目で exec を使って

- (1) まず exec 3<&0 で、標準入力(ファイルディスクリプタ 0) をファイルディスクリプタ 3 に複製する。
- (2) 次に exec 0<<EOF ... で EOF 以下のヒアドキュメントを標準入力として入力する。

とすることで、 while 文中の変数操作をその後でも参照できるように
している。

また、この連想配列を格納している while 文が終わった直後に、

- (3) exec 0<&3 で、(もともとの標準入力を複製していた)ファイルディスクリプタ 3 をファイルディスクリプタ 0 に複製することで、標準入力がもとに戻る。
- (4) 最後に exec 3<&- で使い終わったファイルディスクリプタ 3 を閉じる。

とすることで、その後のスクリプトで影響がないようにしている。

4 trap を用いたシグナル処理

--- 使った一時ファイルを消去 (1) ---

スクリプト中に一時ファイルを作ることはよくある。
そういう場合は、スクリプトが終了するときにその一時ファイルを
削除するようにするが、単にスクリプトの最後で rm しているだけ
では、異常終了した場合(例えば Ctrl-C で中断したとき等)に
その一時ファイルが残ってしまう。
こういう場合には以下のように trap コマンドを使うと便利である。

```
$ cat trap.sh
...
trap 'rm $tempfile; exit' 0 1 2 13 15
...
```

4 trap を用いたシグナル処理

--- 使った一時ファイルを消去 (2) ---

こうしておくと、スクリプト中で $tempfile が作成された後に
異常終了(終了コードが 1,2,13,15)しても(この場合は 0 がある
ので正常終了でも)最後に $tempfile を消去して終わるので、
一時ファイルが残らない。
ただし上の例だと終了コードが 0,1,2,13,15 以外であるときは
trap していないので一時ファイルは残ってしまう。
(ちなみに 9 (KILL)は trap できない)

5 例題スクリプトの解説

--- popcheck.sh の詳細 (1) ---

```
 1 行目  シェルスクリプトを起動
 3 行目  このスクリプトの名前
 4 行目  設定ファイル
 6-- 11  help 用関数
13-- 27  オプション処理
29 行目  (シェルスクリプトのオプションを引数から除去)
30-- 31  ポート,ウエイト時間デフォルト
33-- 68  設定ファイル読み込み or 無かったら exit
70-- 74  パイプ用,プロセス通信用名前つき FIFO 作成
         (trap による終了処理込)
76--133  telnet で popserver に繋いで POP(APOP) プロトコルで通信
         ('|' でパイプを繋いでいるのでサブシェルで実行)
135--238 (名前付き)パイプで繋いで後段処理
         ('{}' のみなのでカレントシェル)
240--254 メッセージ表示数
256--261 サブジェクト表示
263 行目  スクリプトの終了
```

5 例題スクリプトの解説

--- popcheck.sh の詳細 (2) ---

76--133 の {} は '|' でパイプされているので、サブシェルで実行される。後段の telnet も同様。よって変数もそのサブシェル内でのローカル変数である。パイプの後段では telnet の標準出力を名前付きパイプ $pipefile へリダイレクトしておく。

また、パイプのの前段中で until 文で read する際に(実際にはあまり意味はないが)リダイレクトを使用しないようにファイルディスクリプタを操作 (exec 4<&0 0<&{tempfile}) している。

一方 135--238 の {} は前段での telnet の出力をパイプではなくファイルディスクリプタ操作 (exec 3<&0 0<${pipefile}) と名前付きパイプを用いることによって、カレントシェルで実行される。こうすることでこの中の変数はカレントシェルのものであるので、後で値を参照できる。

しかしこの {} 内部でも while 文にリダイレクトやパイプを使うと、この while 文がサブシェルで実行されてしまい、変数のへの値の代入や変更がその内部のみで有効となるので、ファイルディスクリプタを操作 (exec 5>&1 1>&{tempfile}) することによって、 while 文をカレントシェルで実行することにする。

5 例題スクリプトの解説

--- popcheck.sh の詳細 (3) ---

こうすることで、 240--254 の部分で $max を参照でき、また 256--261 の部分で先程のカレントシェルで実行された {} の内部で格納された連想配列を呼び出すことによってサブジェクトを表示できるようになっている。

(単に表示するだけなら {} の内部の閉じたところで echo させてもよいが、こうしておくことによって後々の変更やメンテナンスに有用である。

ただしあまり沢山連想配列を使うとメモリを消費してしまうが、実際にはそれほど気にならないだろう。)

また 132 行目と 237 行目で exec によってそれぞれもとのファイルディスクリプタに戻しておき、その後のスクリプトに影響がないようにしている。

(132 行目はサブシェルで実行されているので別にいらないが一応つけておく)

Acknowledgement

参考にしたもの

- Bourne Shell 自習テキスト
  - http://www.tsden.org/takamiti/shText/
- プロフェッショナルシェルプログラミング
  - アスキー, ISBN4-7561-1632-9
- LILO ML でのシェル関連スレッド

確認に使わせていただいたシェル

- しらいさん作の FD shell
  - http://hp.vector.co.jp/authors/VA012337/soft/fd/
- PDP11 エミュレーション上の UNIX v7 の /bin/sh
- その他 Bourne Shell 系のシェル (bash, zsh, ash, pdksh)

最後に

資料は以下にあります。

- http://www.netfort.gr.jp/~tomokuni/lms/shell/

ご静聴、ありがとうございました。 m(_o_)m

次回(まだやるのか？)もご期待ください。^^;

おしまい。