シェルスクリプト入門

岡野浩三

2023 年 5 月 23 日

1　はじめに —シェルの役割—

例えば LaTeX で文章処理をするときに以下のような一連のコマンドを入力することがある．

```
$platex doc.tex
$platex doc.tex
$dvipdfmx doc.dvi
```

このような 3 つくらいのコマンド系列であれば shell の history 機能を用いて再度入力をするだけでもよいが，それでも複数の異なったコマンドを連続して何回も入力するのは面倒である．可能であればこの一連のコマンドを 1 つのコマンドのようにして例えば以下のように入力することにより先ほどと同じ作業ができるのであればうれしいはずだ．

```
$latex2pdf doc.tex
```

シェルスクリプトはこのように複数のコマンドを組み合わせて一つのコマンドのように扱えるようにする (スクリプトプログラミング言語)．最近ではライトスクリプトとして perl や ruby，python などそのような目的に適したプログラミング言語が流行っているが，それらの元祖ともいえる機能で，UNIX の初期の目論見 (単純なコマンドを組み合わせて複雑な処理をする) を実現する有用な手段であった．また，perl や ruby の構文や変数の使い方などに良くも悪くも影響を与えている．また Windows の wish や powershell(古くは DOS のバッチファイル) にも影響を与えている．

今でも UNIX のいくつかの起動処理などはシェルスクリプトが使われており，UNIX サーバ管理者はシェルスクリプトを少なくとも読み書きできないと仕事にならない．

2　シェルの系列

昔から UNIX 系 OS では少なくとも 2 つの shell の系列が使われてきている．Bourne Shell 系と C Shell 系である．前者はシェルスクリプトとしての機能が重視され，後者は対話的実行が重視されている．

Ubuntu ではユーザーシェルとして bash (Bourne Shell 系) が標準で使われている (ただし近年システムシェルについては dash という新たなシェルが主にセキュリティの観点から使われはじめて

いる．). Bash はシェルスクリプトとしての機能が重視されている一方で対話的実行の機能も強化されていることもあり Linux ではほぼ標準で採用されている．

3　ユーザ実行パスの設定

ユーザが作ったシェルスクリプトは通常のコマンドと同様に使うことができる．再利用しやすくするために，自作のシェルスクリプトや自作の実行バイナリファイルをまとめて置いておくディレクトリを用意し，そのディレクトリに path を設定する．

そして，新しくシェルスクリプトを作成したら，そのファイルを実行できるようにパーミッションを設定する．

そのための手順を以下にまとめる．

3.1　最初に一度だけする手順

1. ユーザーホームの下に実行ファイルを置くディレクトリを用意する．通常は 慣例にしたがい bin ディレクトリを以下のようにホーム直下につくる．当然ながら，すでに存在すれば再度作る必要はない．

   ちなみに Docker では root 権限で起動するのでホームディレクトリは/root になる．

```
$cd ~
$mkdir bin
```

2. ~/.bashrc に以下の記述 (あるいはそれに類した記述) があることを確認する．

```
$more ~/.bashrc

if [ -d $HOME/bin ]; then
PATH="$PATH:$HOME/bin"; export PATH
fi
```

   なければエディタ等を用いて追加しておく．似たような記述があれば大丈夫である．ポイントは`$HOME/bin` が PATH に追加されていることである．.bashrc は bash 起動時毎回に読み込まれ，実行される．

以下を実行すると `$PATH` の内容を確認できる．

```
$echo $PATH
```

古典的には UNIX でよく使われるエディタは vi, emacs などである．vi は標準で使える．

一方，emacs を使おうとしてターミナルで emacs と打つと以下のようなメッセージが出ることがある．

プログラム'emacs'は以下のパッケージで見つかりました:
略
次の操作を試してみてください. : sudo apt-get install <選択したパッケージ>

要するに emacs は標準では Ubuntu に入っていないので追加インストールをする必要がある.
ネットワークが使える環境下で

```
$apt-get update
$apt-get install emacs
```

などと実行するとよいだろう. ただし, かなりの量になる. 演習では Windows がわでファイル編集するのがよいであろう.

vi, emacs の操作の詳細についてはネットにいくらでも資料があるだろう.

nano というエディタはサイズが小さく良いかもしれない.

3.2 シェルスクリプトを 1 つ作るたびに実行する手順

1. シェルスクリプト (たとえば foo) を適当なエディタで記述する.
2. 作ったシェルスクリプトを先のディレクトリ (~/bin) に置く.

```
$mv foo ~/bin
```

3. 作ったシェルスクリプトに実行パーミッションを与える.

```
$chmod u+x ~/bin/foo
```

4. 必要なら次を実行する (.bashrc を再度読み直している).

```
$source ~/.bashrc
```

5. 必要なら次を実行する.

```
$hash -r
```

以上で foo がコマンドのように扱える.

問 1
シェルスクリプトを *1* つ作るたびに実行する手順 *4, 5* はそれぞれどのような場合に実行しないといけないか？

問 2
hash -r の操作の意味役割をしらべよ.

Shell スクリプトはインタプリタ形式で動く．

さきほどの foo コマンドは以下のようにしても動作できる．

```
$bash ~/bin/foo
```

このスタイルのほうがインタプリタ形式の実行方法として自然である (bash にプログラムコード foo を与えてインタプリタ形式で実行している) が shell プログラムの実行のたびに bash をタイプするのは面倒である．上述のようにプログラムコードファイルそのものに実行権限を与える方式が使えるようになっている．

4 スクリプト例

スクリプトの先頭行には以下のおまじないを記述する．

```
#!/bin/bash
```

注意: 互換性や汎用性を重視するなら，ここは

```
#!/bin/sh
```

と記述すべきである．なお，Ubuntu では /bin/sh は実のところ/bin/dash へのハードリンク (同じ実体に別の名前がついている) であり，Ubuntu で/bin/sh を実行すると/bin/dash が実際には実行される．

この資料では互換性がやや下がるが

```
#!/bin/bash
```

で統一することにする．

問 3

上述のおまじないの意味をしらべよ *(ヒント shebang)*.

hello というファイル名で，次のような中身のスクリプトを作る．

注意: 通常の UNIX の作法にしたがうならコマンド (実行可能ファイル) には通常は拡張子はつけない．

```
#!/bin/bash
echo "Hello, world"
```

演習 (レポート報告不要) 1

上述の内容を持つ *hello* をエディタで作成し， */bin* に移動せよ．さらに実行属性をつけなさい．任意のディレクトリ下で *hello* と打ち込んで，作成したシェルスクリプトが動作することを確認せよ．できない場合は「シェルスクリプトを *1* つ作るたびに実行する手順」を確認せよ．

5　コマンドの連続実行

通常は 1 行に 1 つコマンドを書いていく．その順で実行される．1 行に複数のコマンドを記述したいときは次のように記述する．

```
com1 ; com2
```

このように ; で区切ってコマンドの逐次実行を表現できる．この場合，com1 を実行後，com2 を実行する．3 つ以上続けてもよい．ただし，可読性が著しく低下することがあるので多用すべきでない．

```
(com1 ; com2 )
```

() で囲われたコマンド群を「サブシェル」のもとで実行する． サブシェルで行った環境変更はもとのシェルに影響しない．使い方としてはサブシェル内で一時的に作業ディレクトリを変更するということが考えられる．ディレクトリの変更はサブシェルから抜けたときに無効化される (元のディレクトリに戻る).

```
com1 && com2
```

com1 を実行し，成功したならば (正常終了時)，com2 を実行する．com1 の実行が失敗したときは com2 が実行されないことに注意.

コマンドの正常終了か否かの判定は当該コマンドの終了ステータス値で判断する．この終了ステータス値はコマンド側のプログラムでは通常 exit システムコールで行う．なお，シェルスクリプトでも exit コマンドを用いて終了ステータス値を設定できる.

UNIX(Linux) の慣例では，正常終了の値は 0 である.

直前に実行されたコマンドの終了ステータス値は特別なシェル変数 `$?`に格納されるのでそれで確認することもできる.

例 1

`ls > .` はカレントディレクトリに書き込もうとしている．エラー終了コマンドのため値が *1* となっている.

```
$ls
$echo $?
0
$ls > .
$echo $?
1
```

```
com1 || com2
```

com1 を実行し，失敗したとき (正常終了ではないとき) は，com2 を実行する.

6 コメントアウト

から行末までがコメントとして取り扱われる (ただし 1 行目のおまじないを除く).

7 変数

シェルスクリプトの変数 (シェル変数) の説明をする前に，環境変数について説明する．bash では任意のシェル変数 (たとえば FOO) に対して

```
$export FOO
```

を実行すればこの変数 FOO は環境変数になる．

環境変数は設定をしたシェルの子プロセス典型的にはそのシェルから実行するすべてのコマンドなど) から参照することができる．この機構を使うとある種の情報伝達を行うことができる．

少しややこしいのはこのような環境変数のいくつかはすでに役割が決まっていることである．

シェルスクリプト開発手順の最初に一度だけする手順にでてきた `~/.profile` の記述の一部を再掲する．

```
PATH=$PATH:"$HOME/bin;␣export␣PATH"
```

注意: 変数値の設定の際，=の前後にスペースをいれてはいけない．

これは PATH というシェル変数の値を再定義し，export により環境変数であることを宣言しなおしている記述である．この環境変数 PATH はコマンドの検索パスという意味 (役割) を持っている．

今，現在どれだけの環境変数が設定されているかは

```
$printenv
```

を実行すればよい．HOME, PWD, LANG, USER,SHELL など値と名前からおおよそどのような役割があるか想像できるものも多い．

まとめるとシェル変数は export すると環境変数となり，子プロセスから (必要に応じて) 参照できるようになる．いくつかの環境変数はすでに役割が決まっている．環境変数はある種のグローバル変数 (もしくはグローバル定数) のような役割を持つ，ということになる．

逆にいうと，すでに環境変数として使われているものでない変数名については新たな環境変数にすることもできるし，環境変数でない通常のシェル変数として使うこともできる．シェルスクリプトでプログラムのために用いるのは通常のシェル変数である．慣例として通常のシェル変数は小文字を用いる．シェル変数は宣言なしに使える．以下のようにまずは初期値を代入するのが普通であろう．

問 4

ユーザが一時的に値設定したシェル変数を再び無効化するにはどうしたらよいだろうか？

8 変数初期化

変数の宣言と代入が同時に行える．変数名と「=」の間にスペースを入れてはいけない．

```
FOO=abcde
```

上は変数 FOO に値 abcde を設定している．

9 変数参照

$変数名 というように変数値を参照する際には先頭に$を付ける．あるいは，${変数名} のように変数名の範囲を {} でくくると変数名の範囲の曖昧さの問題を回避できる．後者の記述に統一することを推奨する．

10 文字列

文字列は明示的に引用符で引用する．

「'」(シングルクォート) と「"」(ダブルクォート) で引用するのとでは振舞いが異なる．

「'」(シングルクォート) で囲んだ文字列 : 書かれたそのものの文字列

「"」(ダブルクォート) で囲んだ文字列 : $ ` \ の 3 つの特殊文字については変数値 (変数置換), コマンド実行結果 (コマンド置換), エスケープシーケンスとして機能する．

"(ダブルクォート) で囲まれた文字列の中ででは変数の参照ができる．"(ダブルクォート) の囲み内部で上の 3 つの特殊文字を単なる文字として扱いたければ，\ を前につけて \$ \` \\ とする．

注意: 日本語環境の PC では \ は通常 ¥として表示される．

演習 1

先の *hello* スクリプトを次のように改造し実行せよ．エディタの「別名で保存」メニューや *cp* コマンドなどを活用して内容を変えたら別のファイル名にするほうが良いだろう．このとき新規作成したファイルの置き場所や実行属性，事前の *hash* の実行などに気をつけること．以下同様．

```
#!/bin/bash
# Greeting Script

greeting='Hello, world!'
dateInf="today is `date +%m/%d`"

echo ${greeting}, ${dateInf}
```

注意: ` `(逆シングルクオーテーション) と通常の ' ' (シングルクオーテーション) の違いに注意．` `(逆シングルクオーテーション) 内のコマンドが実行されて，その結果の文字列に置き換わる．

問 5
動作内容を説明せよ.

11 引数

シェルスクリプトは引数をともなって実行できる. 引数を処理する際によく用いる特別なシェル変数を以下にまとめる.

変数	意味
`$#`	引数の個数
`$n`	n 番目の引数
	0 番目の引数 `$0` はシェルスクリプト自身の名前
	また, n > 9 である場合は `${12}` などと
	中カッコで囲まないと使えない.
`$*`	引数全て
`${m}-${n}`	m 番目から n 番目までの引数

shift というコマンドを実行すると, `$1` に `$2` の内容が入り, `$2` に `$3` の内容が入り, と引数が一つずつずれる (`$#` は 1 減る). 通常は後述のループとともに使う.

演習 2
welcome というファイル名でスクリプトを次のように作成し実行せよ.

```
#!/bin/bash
# Welcome Script

uid=‘whoami‘
g="$1 says \"Thank you, $uid and $2\""
echo ${g}
shift
g="$1 says \"Thank you, $uid and $2\""
echo ${g}
```

実行時は以下のように 3 つ以上引数を指定すること.

```
$welcome smith alice kate
```

12 入出力

シェルスクリプトが実行中に入出力を行うには次のコマンドを用いる.

コマンド	解説
read 変数	入力
	ユーザからの (キーボード) 入力を待ち，その結果を変数にいれる
echo	出力したいもの
echo -n 出力したいもの	出力
	-n オプションがついていると，出力後に改行しない

例 2

```
#!/bin/bash
echo -n "Please input something : "
read a
echo " Your input is " ${a}
```

13 ヒアドキュメント

テキスト内に入力を書くこともできる.

例 3
END が区切りの識別子となっている *(*このキーワードは変更できる*)*.

```
cat <<END
This sentence will be
input through
standard input stream
END
```

演習 3
hello スクリプトをさらに次のように改造せよ．そして，このスクリプトを実行してみるとともに解説せよ.

```
#!/bin/bash
# Greeting Script
g='Hello'
i='Please tell me your name: '
o="Oh, nice to meet you"

echo ${g}.
```

```
echo -n ${i}
read ans
echo ${o}, ${ans}
```

14 計算

オリジナルの sh は原則として計算機能を持っていない (bash 自体は実際には計算機能を持っている).

古典的にはシェルスクリプト中で計算を行いたいときは外部コマンド (expr, bc) を呼び出す手法を使う.

例 4

expr を使った例 *(整数)*

```
#!/bin/bash
a=5
b=`expr ${a} + 10` ; echo ${b}
c=`expr ${b} \* 2` ; echo ${c}
```

‘ ‘ の中では $\star$ は展開されてファイル名などになってしまいかねないので, \ で単なる文字としておく.

注意: ‘ ‘(逆シングルクオーテーション) と通常の ’ ’ (シングルクオーテーション) の違いに注意

‘ ‘(逆シングルクオーテーション) 内のコマンドが実行されて，その結果の文字列に置き換わる.

数字と演算子の間に空白を１つ以上入れる必要がある．expr はトークン（数字や演算子）を引数単位で受取ることを仮定している.

演習 4

上のスクリプトを試してみよ.

例 5

bc を使った例 *(実数も扱える)*

bc は，標準入力から計算式をもらうことで計算を行う機能があるので， echo でリダイレクトすることでそれを利用する.

なお，docker で使用する場合事前に bc をインストールする必要がある.

```
$apt-get update
$apt-get install bc
```

また必要なら docker commit を実行しておくとよい.

```
#!/bin/bash
a=‘echo "4*a(1.0)" | bc -l‘ ; echo ${a}
b=‘echo "c($a)" | bc -l‘ ; echo ${b}
```

演習 5
上のスクリプトを試してみよ．何を計算しているだろうか?

15 シェルの関数

シェル内で関数 (手続き) を作成・利用することができる．

```
functionName () {
    commands …
}
```

関数に対する n 番目の引数を `${n}` として参照できる．この関数内部だけで使いたい変数があるときは変数宣言の時に local を頭部につける．

```
#!/bin/bash

mawaru () {
  local i=1
  while [ $i -le 10 ]
  do
    echo $i
    i=‘expr $i + 1‘
  done
  }

mawaru
```

問 6
このシェルスクリプトはどういう動作をするだろうか?

演習 6
hello スクリプトをさらに改造して次のようにせよ．

```
#!/bin/bash
# Greeting Script
```

```
g='Hello'
d=`date +%d`
dm=`expr $d - 1`
t=`date +%H`
m=`date +%M`

d2min () {
min=`echo "(($1 * 24) + $2) * 60 + $3" | bc -l`
}

d2min ${dm} ${t} ${m}
echo ${g}, "about $min minutes have passed this month."
```

16 制御構造

if 条件分岐

```
if cond1
then com1 …
elif cond2
then com2 …以下繰り返し

()

else comN …
fi
```

制御フローの解説は不要であろう．注意として条件部記述 (cond1, cond2 など) も実はコマンドを書くということである．正常に実行されたコマンドは終了ステータスとして 0 を返すことを思い出してほしい．0(すなわち真) であれば then の後のコマンドが実行され，それでなければ elif もしくは else 以降のコマンドが実行される．

条件部として普通の条件判定をしたいときのためにコマンド test がある．詳細は条件式および条件式のチェックの節を参照のこと．

演習 7

次のスクリプトを作成し，実行せよ．

if のすぐ後の `[ -e $fn ]` という部分については条件式および条件式のチェックの節を参照のこと．

```
#!/bin/bash

fn=~/.bashrc

if [ -e ${fn} ]
   then
   cat ${fn}
   else
   echo "You␣has␣no␣${fn}."
fi
```

17 Case(パターンマッチ) 条件分岐

```
case 文字列 in
    pattern1 ) …com1 ;;
    pattern2 ) …com2 ;;以下繰り返し

()

    * ) …comN
esac
```

最初に 文字列 を評価し， 文字列が pattern1 とマッチする場合 com1… が実行されて終了する．マッチしない場合次のパターンとマッチするか調べる．いずれにもマッチしない場合* ）の comN が実行されて終了する． *）の項は存在しなくても良い．

演習 8
次のスクリプトを作成し，実行せよ．

```
#!/bin/bash

echo -n "Please␣input␣file␣name:␣"
read fn

case ${fn} in
    *.txt )
        echo "${fn}␣is␣text␣file.";;
    *.c )
```

```
        echo "${fn} is C program file.";;
    *)
        echo "Hey, what is ${fn} ? Do you know it?";;
esac
```

18 for 変数を用いた繰り返し

```
for Variable in PatternorList
do
    commands
done
```

まず，PatternorList を展開し，リストを生成する．そのリストの要素を順番に Variable に代入して，そのたびに commands を実行する．

演習 9
次のスクリプトを作成し，実行せよ．

```
#!/bin/bash

for x in 0 1 2 3 4 5 6 7 8 9 10
do
    v=`echo "${x} * 0.31416" | bc -l`
    w=`echo "s(${v})" | bc -l`
    echo "sin(${v}) = ${w}."
done
```

19 while, until 条件繰り返し

```
while Condition
do…
    commands
done
```

until の構文も同じ．
Condition が成り立つかチェックする．
[while の場合] 成り立つならば，commands を実行．
[until の場合] 成り立たないならば，commands を実行．
再び条件が成り立つかチェックし繰り返す．

演習 10
次のスクリプトを作成し，実行せよ．

```
#!/bin/bash
x=0
while [ ${x} -le 10 ]
do
    v=`echo "${x} * 0.31416" | bc -l`
    echo "sin(${v}) = " `echo "s(${v})" | bc -l`.
    x=`expr ${x} + 1`
done
```

20 break … ループ脱出

for，while, until のループから脱出する．

exit … 終了．シェルスクリプトの終了．正常終了時は 0 を渡し，エラー時はそれ以外の値を渡すべきである．

21 条件式および条件式のチェック

sh 系シェルでは，条件は基本的に [条件式] という形式で書かれる．

" [" と条件式の間にはスペースが必要なので注意すること．

[は条件式をチェックする，という機能を持つコマンドである．OS によっては test と [とはリンク (ひとつのファイル実体に対し，異なる名前を付けること) されている．上述のスペースの注意は結局のところ条件式の各項がこのコマンドのオプション引数として解釈されることを意味している．調べたところ残念ながら Ubuntu では [と test は別のコマンドである．

22 シグナル割り込み

UNIX のプロセス間でシグナルによる割り込み処理 (簡単な通信機能) が使える．CPU による割り込み (命令) とは別のもので，OS が提供する機能である．ただし，考え方は CPU の割り込み命令とほぼ同じであるし，用語も同じである．しかも実装には結局のところシステムコールを最終的に使うわけであり，まったくの無関係というわけではない．ただし，混同しないように気をつけよう．

Linux UNIX のプロセス間通信で提供されているシグナルは 15 種類以上ある．ここではその中でよく使われる，SIGINT, SIGTERM だけ使ったシェルスクリプトを説明する．

一般に CPU の割り込み処理ではプログラムに以下の記述をする．

- 割り込み処理の記述 (割り込みハンドラ)

- 割り込みハンドラと割り込み信号との対応付け (割り込みベクトル)
- そしてもちろん，通常の処理

繰り返しの説明になるが CPU の通常の処理の途中で割り込み信号が発生した場合，通常処理を一旦中断し，その割り込み信号に対応する割り込みハンドラを割り込みベクトルから探し，対応する割り込みハンドラの処理 (通常はきわめて短い) をおこない，終了しだい，iret など割り込みハンドラから復帰する命令が実行され，もとの通常処理を中断したところから再開する．

UNIX のプロセスレベルでは信号による割り込みは以下のように考える．割り込みの発生:ユーザやプロセスが行う．あるいは kill コマンドを用いる．割り込みハンドラや割り込みベクトルの記述:シェルプログラムでは trap コマンドを用いる (名前こそ同じだが，CPU の trap 命令ではなく trap という名前のコマンドである).

例 6
ユーザがキーボードから *(*端末から*)* Ctl+C を押せば，その端末から起動し現在動いている全プロセスに一斉に *SIGINT* が送られる．その信号を受けた各プロセスがどのように振舞うかはそのプロセスのプログラムにおける「その信号をどのように処理するか」の記述に従う．プログラマが明示的に処理を記述していない場合の *SIGINT* に対するデフォルトの動作はプロセスの終了となる．これはシェルスクリプトでも同様．このおかげで通常は Ctl+C を押すことによりプログラム *(*プロセス*)* を終了させることができる．

kill コマンドでオプションなしに指定プロセス ID に実行する (たとえば kill 123) と指定プロセス ID(123) のプロセスに SIGTERM 信号が送られる．プログラマが明示的に処理を記述していない場合の SIGTERM に対するデフォルトの動作もプロセスの終了となる．

一方 trap コマンドの書式は以下のとおり．

```
trap comand singal ···
```

最初に簡単なプログラム例を提示する．

```
#!/bin/bash

trap 'echo ======= I am trapped. =======' SIGINT
x=1
while true  # infinite loop!
do
    echo "${x}-th loop!!"
    x=`expr ${x} + 1`
done
```

trap コマンドの実行の意味はこうである．SIGINT のシグナルがきたときは echo コマンドを実行し，指定の文字列を実行する．そして，それ以降に通常のプログラムが記述されている．中身は読めば理解できるように無限ループである．少し危険なにおいを感じていただけるだろうか？

実行の際，以下のように必ず terminal を 2 つ以上起動させておくこと．

Windows Terminal で 2 つ目のターミナルを Ubuntu として起動する．

上述のプログラムをファイル名 e13 で作成したとする．1 つ目の terminal で実行しよう．

```
$ ./e13
```

想像どおり，無限ループに突入する．ここで終了のため Ctl+C を何度押しても先ほどの trap コマンドで動作を書き換えてしまったため

```
======= I am trapped. =======
```

が Ctl+C を押したタイミングで表示されるだけで，終了されない．

落ち着いて先ほど開いていた別のターミナルにフォーカスをあて，ここから操作しよう．まず，

```
$ps a | grep e13
```

と打つ． ps に a オプションを指定することを忘れずに．また e13 は先ほど起動したスクリプト名である．適当に読み替えること．たとえば次のような表示が得られる．

```
3464 pts/0    R+     0:02 /bin/bash ./e13
25032 pts/4    S+     0:00 grep --color=auto e13
```

この例では e13 のプロセス id が 3464 であることが確認できる．あとは kill コマンドでこのプロセスを終了すればよいだけである．

それをする前に kill コマンドが本来任意の信号を発生できることを思い出して，以下で遊んでみよう．

```
$kill -INT 3464
```

これでプロセス ID 3464 すなわち現在無限ループで暴走？しているスクリプトに対して SIGINT シグナルを出していることになる．先ほどと同じ反応をするはずだ．

別のターミナルから遠隔操作しているようでなかなか面白いのではないだろうか．本当に終了したい場合は

```
$kill 3464
```

とすればよい．デフォルトのシグナルは SIGTERM であり，これの挙動は変えていないので無事終了するはずである．

注意: 以下，Docker の場合です．

落ち着いて先ほど開いていた別のターミナルにフォーカスをあて，ここから操作しよう．まず，

```
$ps a
```

と打つ． ps に a オプションを指定することを忘れずに．たとえば次のような表示が得られる．

```
3464 pts/0    R+     0:02 /bin/bash
25032 pts/2    S+     0:00 ps
```

pts/0, pts/2 はターミナル名を意味する．ps コマンドを実行したターミナルが pts/2 で　スクリプトを実行したターミナルが pts/0 であることが想像できる．

本来であれば/bin/bash ではなくスクリプトのファイル名である e13 などが表示されるのであるが，Docker ではファイル名ではなく bash というプロセス名で表示されている可能性がある．場合によっては　無事 e13 というプロセスが表示されている．

いずれにせよ動作中のプロセス id が 3464 であることが確認できる．あとは kill コマンドでこのプロセスを終了すればよいだけである．

それをする前に kill コマンドが本来任意の信号を発生できることを思い出して，以下で遊んでみよう．

```
$kill -INT 3464
```

これでプロセス ID 3464 すなわち現在無限ループで暴走？しているスクリプトに対して SIGINT シグナルを出していることになる．先ほどと同じ反応をするはずだ．

別のターミナルから遠隔操作しているようでなかなか面白いのではないだろうか．本当に終了したい場合は

```
$kill 3464
```

とすればよい．デフォルトのシグナルは SIGTERM であり，これの挙動は変えていないので無事終了するはずである．

このままでは少し厄介なので次のようにプログラムを書き換えてみる．

```
#!/bin/bash

trap 'echo ␣=======␣I␣am␣trapped.␣=======;␣trap␣SIGINT' SIGINT
x=1
while true  # infinite loop!
do
    echo "${x}-th␣loop!!"
    x=`expr ${x} + 1`
done
```

変更部は trap コマンドで実行する内容である．前回と同様に echo で文字列を表示させたあと，再度 trap SIGINT を実行している．今度は trap 時の実行コマンドを指定していないことに注意．

この記述の場合はシグナル受け取り時の動作をデフォルトにリセットすることを意味している．

したがって，このスクリプトプログラムの挙動は 1 回目の INT(すなわち Ctl+C) では文字列を表示したのちループを再開し，2 回目の INT では無事終了するはずである．

次に SIGTERM の挙動を変えてみよう．

```
#!/bin/bash

trap 'echo ␣=======␣I␣am␣trapped.␣=======;␣trap␣SIGINT' SIGINT
trap 'echo ␣=======␣I␣will␣finish␣the␣loop.␣=======;␣exit␣␣1' SIGTERM
x=1
while true  # infinite loop!
do
    echo "${x}-th␣loop!!"
    x=`expr ${x} + 1`
done
```

このようにシグナルごと別の挙動を与えることができる (これらの記述はちょうど割り込みハンドラと割り込みベクトルの関係みたいではないですか).

SIGTERM のシグナルがきたときは

```
======= I will finish the loop. =======
```

と表示した後，終了する (exit 1).

さきほどと同様に別ターミナルから kill コマンドでシグナルを送ってみよ.

この例では echo で文字列を表示させるだけのプログラムだったが実際のプログラムでは作業の後始末などに使える.

たとえば以下のプログラムは実行中に一時ファイルを作成し，最新の出力をそのファイルに保存している. Ctl+C や SIGTERM が発生した場合は，終了の手前でその一時ファイルを削除している (後始末をきちんとしている).

```
#!/bin/bash

trap '/bin/rm␣tmp.$$;␣exit␣1' SIGINT SIGTERM

x=1
while true  # infinite loop!
do
    echo "${x}-th␣loop!!"
    echo "${x}-th␣loop!!" > tmp.$$
    x=`expr ${x} + 1`
done
```

このような処理が本来の正当な使い方であろう. ちなみに $$ は現プロセスの pid を意味する変数である. 複数プロセスが起動したときに tmp ファイルを pid を区別するときに使える.

演習 11

実際に動かしてとめてみよ. くれぐれも *terminal* を *2* つ以上確保することを忘れずに.

演習 12
expr の代わりに *bash* の組込みの計算機能を使う方法を調べ，それを使うように改変せよ．また *bash* の組込みの計算機能を使う利点，欠点をあげよ．

演習 13
trap 時の動作内容を関数として与えるようにプログラムを変更せよ．*trap* 時の動作として複雑な動作を指定するのであれば，それらの内容を関数として定義して与えるほうが可読性が向上する．

演習 14
ループ時の処理を *1* 秒ごと繰り返すように改変せよ．

少し禁断の領域？に踏み込んでみる．

```
#!/bin/bash

trap 'echo ==== I am trapped ====' SIGINT SIGTERM

x=1
while true  # infinite loop!
do
    echo "${x}-th loop!!"
    x=`expr ${x} + 1`
done
```

このスクリプトを読んで危険なにおいがわかるだろうか
前回は Ctl+C が使えなかった代わりに SIGTERM では終了したので 別ターミナルから kill コマンドを用いて事なきを得た．
今度のプログラムでは SIGTERM の挙動も終了できない処理に変えてしまっている．このプログラムを動かしたのち，とめるにはどうしたらよいだろうか？

演習 15
実際に動かしてとめてみよ．くれぐれも *terminal* を *2* つ以上確保することを忘れずに．ヒント *SIGKILL*

このヒントで答がわかってしまった人に追加問題

演習 16
SIGKILL を上述と同じように他の *signal* とまとめて *trap* したらどうなるか？

問 7
while のあとの条件部に書かれている *true* は何者だろうか? 条件部はコマンドでなくてはいけなかった．

演習 17
以下の要件をみたすオリジナルのスクリプトを作成し，そのスクリプトの解説，実行の様子などをまとめよ．

- パイプを用いること
- *SIGINT* に反応し，反応時には中断以外の独自の動作をすること
- 全体でまとまった意味のある動作をすること

演習 18
画像ファイルのタグ付けシステム．スマートフォンでは画像に対して複数のタグを付けることができる．この機構をファイルの *link* 機構を用いて実現してみよう．

環境変数 *IMAGEFOLDER* に実在するディレクトリが指定されていると仮定する *(*例えば */home-/yourname/images)*．さらに*$IMAGEFOLDER/base* ディレクトリ下に *jpg* ファイルが画像として格納されていると仮定する．

- コマンド *imgls* で *$IMAGEFOLDER/base* 下のすべての画像ファイルのファイル名を１行ごと列挙するようにしたい．*imgls* をシェルスクリプトとして実現せよ
- コマンド *imgtag* タグ名 画像ファイル名で画像ファイル名に タグ名のタグを付ける．画像ファイル名は*$IMAGEFOLDER/base* 下に存在すると仮定してよい．タグを付ける実装は*$IMAGE-FOLDER/*タグ名のディレクトリをなければ作成し*$IMAGEFOLDER/*タグ名*/*画像ファイル名を*$IMAGEFOLDER/base/*画像ファイル名へのリンクとする．すでにリンクがあるときは何もしない．指定された画像ファイルが存在しない場合は，適切なエラーメッセージを出力すること．コマンド *imgtag* をシェルスクリプトとして実現せよ．
- コマンド *imgls* タグ名でタグ名以下ののすべての画像ファイルのファイル名を１行ごと列挙するようにしたい．コマンド *imgls* を拡張実装せよ．
- コマンド *imgls -tagname* でタグ名の全リストを１行ごと列挙するようにしたい．コマンド *imgls* を拡張実装せよ．

すでにタグのリンクがあるときはなにもしないと，上述しているが，リンクの有無を *if* 等で条件判定し，適切な処理をするように記述すること．

23 参考

ファイルに関する条件式

-e 名前　その名前のファイルが存在すれば真
-d 名前　その名前のディレクトリが存在すれば真
-f 名前　その名前の通常ファイルが存在すれば真
-r 名前　その名前の読み込み可能なファイルが存在すれば真
-w 名前　その名前の書き込み可能なファイルが存在すれば真
-x 名前　その名前の実行可能なファイルが存在すれば真
-s 名前　その名前の (サイズが 0 よりおおきい) ファイルが存在すれば真

文字列に関する条件式

-z 文字列　文字列の長さがゼロならば真
-n 文字列　文字列の長さがゼロでなければ真
文字列 1 ==　文字列 2 (= 2 つ) 文字列が等しければ真
文字列 1 !=　文字列 2 文字列が異なれば真

数値に関する条件式シェルは条件式の判定には整数しか用いることができない.

数値 1 -eq 数値 2　数値が等しければ真
数値 1 -ne 数値 2　数値が異なれば真
数値 1 -gt 数値 2　数値 1 $>$ 数値 2 ならば真
数値 1 -ge 数値 2　数値 1 $\geq$ 数値 2 ならば真
数値 1 -lt 数値 2　数値 1 $<$ 数値 2 ならば真
数値 1 -le 数値 2　数値 1 $\leq$ 数値 2 ならば真

その他の条件式

!条件式　NOT
条件式 1 -a 条件式 2　AND
条件式 1 -o 条件式 2　OR
(条件式)　条件式をグループ化する.