シェル・ブートキャンプ

～シェルを使った作業効率アップの秘訣～

Ver. 1.0

リナックスアカデミー矢越昭仁

2012/01/28

Linux のシェルは単なるコマンドインタプリタとしてだけではなくプログラミング言語としても十分な機能があります。この講座ではコマンドラインから効率的な処理を行う方法と、実務で使える簡単なシェルスクリプトの事例を交えながら紹介します。

目次

はじめに

シェルの概要

シェルはキーボードからユーザの指示をうけ、種々のプログラムを起動しその結果をファイルに吐き出したり、他のプログラムを停止させるなどの処理を行います。キーボードから文字を入力して操作することから、このようなコンピュータの利用形態は CUI（Command / Character User Interface)と呼ばれます。UNIX/Linux ではシェルと呼びますが、他の OS ではコンソールコマンド、コマンドプロセッサなどと呼びます。
GUIは操作性が高くユーザにとってわかり易いと一般に言われますが、定型的な作業や繰り返し実行などは CUI の方が優れています。またプログラムに組み込む事でシステムを自動制御するといった事も実現できるため、Windows でもシェル機能は年々強化されているようです。

複数のシェルがあり選択できるのは UNIX/Linux の特徴で、主な物は以下のとおり。

- sh (Bourne shell)、B-Shell
  AT&T UNIX Ver.7 で採用されたシェル。名称は作者であるスティーブ・ボーンにちなむ。他の OS のコマンド言語と違い、プログラム言語としても使えるよう意識して設計されたこと、UNIX ver7 に標準搭載されたことなどから、多く広まった。事実上の UNIX 標準。

- csh、C-shell
  4.1BSD版開発時に採用されたシェル。文法がC言語に似せて作られていることからその名がついた。ジョブコントロール機能が追加され、ジョブの一時停止、再開、バックグラウンド・フォアグランド切り替えなどができるようになった。BSD 系での標準。

- tcsh(tenex csh)
  csh を拡張し、ユーザインタフェース部分を向上させたシェル。作者が当時携わっていた人工知能プロジェクトで使用した PDP-10(DEC) /TENEX に触発されて作られたといわれる。コマンド補完機能は TENEX のもの。

- ksh(Kone shell)
  AT&T のデイビッド・コーンが作者であり、B-shell を機能強化し csh のジョブコントロール機能や、tcsh のコマンド補完機能などを取り込む。POSIX[1]の標準(ksh88)として採用され、2000 年以降はフリーとして提供さている。

- bash(Bourne-Again shell)
  GNU が sh の上位互換として新たに開発したシェルで Linux の標準となっている。名前は当時人気が下火になっていた B-shell を再び世に知らしめるためと言われている。そのため文法的には B-shell を踏襲している。

- zsh(Z shell)
  究極のという意味をこめ 'Z' shell と名づけられており、過去に登場したシェルの機能を取り込んでいる。コマンド補完時の簡単なヘルプを表示、入力行が長くなった場合に見やすくスクロールするなど、特にユーザインタフェースが改良されている。

この様にシェルには沢山の種類がありますが、基本的には B-shell 系と C-Shell 系の 2 系統になります。この資料では Linux で標準となっている Bash を基本として解説します。

補足）
CentOS5.7 における、パッケージとシェルの関係は、bash-3.2-(sh, bash)、tcsh-6.14(csh,tcsh)、ksh-20100201(ksh)、zsh-4.2(zsh)となっています。

---

[1] Portable Operating System Interface X, 1990 年代に乱立した商用 UNIX の API を統一する目的で米政府により制定された規格。現在は IEEE 1003.x、ISO/IEC JTC 1/SC 22 の国際規格となっている。

シェルの仕組み

プロセスの生成

OS でプログラムを動かすには、プログラム本体とそれを実行する上で必要となる入出力装置、CPU、メモリといったコンピュータ資源を割り当てる必要があります。プログラム本体はバイナリ、実行可能形式、エクゼなどと呼ばれ、ほとんどの OS はファイルとして用意されています。そのファイルを OS に読み込ませ、入出力装置と必要なメモリや CPU を割り当て実行させることがプロセスの起動です。

汎用機やレガシーシステムと呼ばれるコンピュータでは、この起動に関する部分を専用の言語である JCL(Job Control Language)で記述します。JCL では CPU の割り当て時間、メモリ使用量、出力ファイルはその容量、増分、上限といった事まで細かく管理する事ができます。

```
//NIGHT01 JOB (NIGHT01S01),'DB CLOSE',CLASS=B,MSGCLASS=X
//STEP01   EXEC PGM=DBB01001
//SYSPRINT DD  SYSOUT=*
//INPUT    DD  DSN=DB01S002.INPUT,DISP=SHR
//OUTPUT   DD  DSN=DAY01.OUTPUT,
//         DISP=(NEW,CATLG,DELETE),
//         SPACE=(CYL,(40,5),RLSE),
//         DCB=(RECFM=FB,LRECL=115,BLKSIZE=0),
//         VOL=SER=VOL001
//
```

プロセスの生成は多くの OS でとても手間のかかる処理です。そのため複数のプログラムを効率的に実行させるためには、事前にそのプログラムが利用するコンピュータ資源を予約する必用がありました。先の JCL はこの発想に基づき、資源を定義しています。またメモリの割り当て量や CPU の優先順位はジョブクラスという小分けにされた実行環境に紐づいていて、プログラムはその中で順番に実行されます。

UNIX/Linux では事細かに資源を指定する事はなく、プログラム名を指定し起動させるだけです。入出力ファイルの指定も引数やリダイレクションで行います。

```
$ dbclose  input.dat  > day01.out
```

UNIX/Linux はシンプルで OS の起動時に init と呼ばれるプロセスを1つ生成するだけです。それ以外のプロセスは、この実行環境である init をコピーし、中身のプログラムを入れ替え、周りの資源を調整するという方法を使っています。多少乱暴な方法ですが、プロセスの生成が簡単で素早く行う事ができます。このプロセスの分身を作って、中身を入れ替える方法を UNIX/Linux では fork(2)と呼んでいます。具体的には以下のようになります。

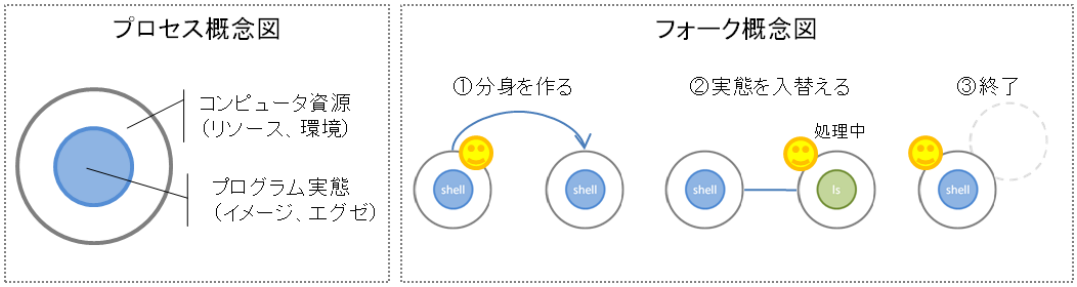


図 1:プロセス処理概念図

たとえば、キーボードから ls [Enter] と入力されると、

1. 自分の分身を作る
   まずシェルが自分自身の分身を作ります。そこには今まで使っていた入出力装置、確保したメモリ領域、その他環境(今いるディレクトリ、ユーザなど)が引き継がれています。(fork(2))
2. 実態を入替える
   予め設定されている PATH 変数の内容に従い、ls という実行ファイルを探します。そして、最初に見つかった実行可能ファイルを読みだし、プロセスの中身を入れ替えます。(exec(2))

あとは、この分身が処理を継続し、その間は親プロセスは休止します。

3. 終了
   コマンドの処理が終わったら、不要なメモリは解放し、一次作業ファイルは削除するなど後始末をします。すべての後片付けが終わったら、親に制御を戻します。(_exit(2))

```
$ ls
Readme.txt  a.zsh    data1.txt  docs     man1len.lis    usrbin.lis
a.png       bin      data2.txt  join.awk  samba.xls       x.cpio
a.tif       bin.lis  Desktop    kits      src             x.so

$ echo $PATH
/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/home/student/bin

$ ls -l /usr/kerberos/bin/ls
/bin/ls: /usr/kerberos/bin/ls: No such file or directory
$ ls -l /usr/local/bin/ls
/bin/ls: /usr/local/bin/ls: No such file or directory
$ ls -l /bin/ls
-rwxr-xr-x 1 root root 95116 Jul 22 10:17 /bin/ls
```

特殊文字と評価

シェルには特別な意味を持つ文字があり、シェルが何らかの処理を行います。

- 「区切り文字(Delimiter)」または「メタキャラクタ (Meta Character)」
  入力された文字列を単語に分割する文字。これらによって分割された文字列の、先頭の単語がコマンド(実行ファイル名)、残りが引数となります。

- パラメータ展開文字
  コマンドライン上で文脈から複数の引数や文字列に展開される文字です。

- パス名展開文字
  展開文字の一種で存在するパス名に展開されます。機能が豊富なので別途記載しています。

種別	文字	意味
区切り文字	| (縦棒)	コマンドの出力と入力を連結し同時実行(パイプライン処理)
	&(アンパサンド)	コマンドのバックグラウンド実行
	;(セミコロン)	コマンドの逐次実行
	(～)	サブシェル(別環境)での実行
	<, >	入出力の切り替え(リダイレクション)
	空白、[Tab]	コマンド、引数の区切り
パラメータ展開	$名前	変数の値を参照。{}や()を伴い、より複雑な操作が可能
	${名前#キーワード}	初期値、文字列変換、文字数などの文字列処理
	$(コマンド、式)	コマンドの実行結果、演算などの処理
	‘,”,` (クオーテーション)	展開の制限、詳しくは後述。
パス名展開	{～} (ブレース)	要素をカンマで区切り、複数のパス名を表現 ex)　$ mkdir　/work/www/{html,cgi-bin,images}
	~ユーザ (チルダ)	指定ユーザのホームディレクトリ、省略時は自分自身 ex)　# mkdir ~student/public_html
	*	任意の文字列が続く(0 個以上)。ex)　ls　/etc/hosts*
	?	何か 1 文字。ex)　echo /usr/bin/?
	[n]	カッコ内のいずれか 1 文字。n-m として範囲指定も可能 ex)　ls　/etc/rc[235].d

展開などの処理はシェルが行う所が他の OS と大きく違います。この事を「評価(eval)」と呼びます。
さらにシェルスはプログラミング言語として、条件判定や繰り返しといった処理を行う事ができます。

コマンド処理(基本操作)

まずは普段システムを利用する際に、コマンドによる指示を効率よく行うポイントを紹介します。利用するシェルは全て bash で、CentOS 5.7, Bash 3.2 で動作確認をしています。

コマンド補完

コマンド名やファイル名は、全て入力しなくとも途中まで入力すれば、残りを bash が継ぎ足してくれます。これをコマンド保管機能(Command compleion)と呼び、tcsh から実装されています。

・コマンド名の補完
プロンプトに入力する途中で、[TAB]キーを押すと入力した文字で始まるコマンドが表示されます。複数存在する場合は、[TAB]キーを続けて2回押せば一覧が表示され、その中から特定できるまで文字を追入力すれば補完されます。

```
$ sha5[TAB]            –保管される→      $ sha5sum
$ sha2[TAB][TAB]       （候補の表示）更に 2 を追加  $ sha22[TAB]    →  $sha224sum
                sha224sum  shaa245hmac   sha256sum
```

なおコマンド名の置換は環境変数 $PATH に示されるディレクトリを検索し、合致した実行可能ファイルが採用されます。

・ファイル名の補完
引数に指定するファイル名も、コマンド名と同様の操作で保管する事ができます。この時、大正がディレクトリであれば、末尾にスラッシュ(/)が付与されます。

```
$ ls -l   /*/*hosts*[TAB][TAB]
ghostscript/  hosts   hosts.allow     hosts.deny
```

この例では、ghostscript がディレクトリである事を示しています（/で終了）。またディレクトリを特定せず、2 階層目を指定している点も知っておくと便利です(/*/~ の部分)。

この様にコマンド補完機能を使えば、簡単な Typo(打ち間違い)によるミスは少なくなります。

コマンド履歴

過去に入力したコマンドを呼び戻す機能をヒストリと呼びます。遡るには上矢印([↑]、Ctl+P)、進むには下矢印([↓]、Ctl+N)を押し、その場で修正する事もできます。
修正したコマンドは、行の途中でも[Enter]を押せば実行できます。行末へ移動する必要はありません。

```
$  vi   /etc/sysconfig/network-scripts/ifcfg-eth0
  ↑Ctl+A                                          ↑Ctl+E
```

カーソルが行の途中（この場合'o'）にあっても[Enter]を押せば、当該のファイルを編集する。
また、行の先頭へは Ctl+A、行末へは Ctl+E でカーソル移動が可能。コマンドラインでのこうした機能は readline と呼ばれる。詳しくは bash のマニュアルを参照のこと。

記録されている履歴を表示するには history コマンドを用います。ヒストリは番号で管理されており、一覧の中から任意の番号を実行するには、!<番号>を用います。

```
$ history
   10  date
   11  netstat -nr
   12  ping 192.168.182.2
$ !11
netstat -nr
Kernel IP routing table
Destination   Gateway         Genmask         Flags   MSS Window  irtt Iface
192.168.182.0 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         0 0          0 eth0
0.0.0.0               192.168.182.2   0.0.0.0 UG        0 0          0 eth0
```

履歴参照ではさらに、!<文字列>で指定した文字列で始まる最初の履歴を再実行、!!で直前に実行したコマンドの再実行ができます。また実行せずにそのコマンドを表示する場合は !～:p のように、最後にコロンと p を付けます。これらはイベント指示子と呼ばれます。

表 1: 主なイベント指示子

イベント指示子	概要
!!	直前のコマンドを再実行
!n	履歴 n 番目を実行
!-n	履歴 n 個分遡ったものを実行
!<文字列>	文字列で始まる履歴の再実行
^文字列 1^文字列 2^	文字列を変換して直前のコマンド再実行
!?<文字列>	文字列を含む直近のコマンドを再実行

ヒストリはファイルに保存し、次回ログインした際にその内容を引き継ぐことができます。保持する履歴の数、ファイル名は環境変数で変更する事ができます。

表 2:ヒストリに関する環境変数

変数名	機能	既定値
HISTSIZE	ログイン中の保持履歴数	500
HISTFILE	履歴記録ファイル名	~/.bash_history
HISTFILESIZE	履歴記録ファイルの件数	500

エイリアス

良く使うコマンドやオプションの組み合わせは、エイリアスとして登録しておくと便利です。CentOS では規定値として以下のエイリアスが定義されています。

表 3:エイリアスの既定値

エイリアス	展開後	解説
ls	/bin/ls --color=tty	ls 実行時に色をつける
l.	ls –d .* --color=tty	カレントディレクトリのドットで始まるファイルの一覧
ll	ls –l --color=tty	詳細なファイル一覧
which	alias | /usr/bin/which --read-alias --tty-only --show-dot --show-tilde	which(コマンドファイルののフルパス表示)に alias の一覧を取り込む。
cp	cp –i	コピー上書き時に確認(root のみ)
mv	mv –i	移動上書き時に確認(root のみ)
rm	rm –i	削除時に確認(root のみ)

上記の which のように、単なるオプションの指定だけでなく、簡単なパイプラインを含ませる事ができます。エイリアスがコマンドと同名の場合は、エイリアスが優先されます。古い C-shell などは、コマンド名と同じエイリアスがあると無限ループしてしまうため、コマンド名にはフルパス名を指定する必要があります。
エイリアスの定義は alias コマンドで行う事ができます。削除は unalias で行います。

```
$ alias di='df -ih'
$ di
Filesystem          Inodes   IUsed   IFree IUse% Mounted on
/dev/sda2             4.5M     98K     4.4M    3% /
/dev/sda1              75K      35      75K    1% /boot
tmpfs                 127K       1     127K    1% /dev/shm
$ unalias di
$ di
bash: di: command not found
```

引数なしの aliasコマンドで、aliasの一覧が表示されます。なお aliasは C-shellで実装された機能で、B-shell では同様のことを関数で行います。B-shell の関数定義には特にコマンドはなく、中小カッコを使って記述します。

```
$ di()
> {
> df -ih
> }
$ di
Filesystem         Inodes   IUsed   IFree IUse% Mounted on
/dev/sda2             4.5M     98K     4.4M    3% /
/dev/sda1              75K      35      75K    1% /boot
tmpfs                 127K       1     127K    1% /dev/shm
$ declare -f
di ()
{
    df -ih
}
$ unset di
$ di
bash: di: command not found
```

なおシェルは入力した文字列が、指示として終了していないと判断した場合は、継続プロンプト(上記例では> 、PS2 により変更可能)を表示し、[Enter]を押しても続きを求めます。

関数は変数として記憶されるため、削除は unset を用います。定義内容の一覧は setコマンドまたは、ksh で導入された declare コマンドの f(function、関数)オプションで確認します。この例では()の後ろには{}が必要であること、一度 { を入力したら、それに対応する } が必要な事から継続しています。

この様に bash では、B-shell(K-shell)と C-shell の両方の機能を持つため、同じ事を違う表現で表わす場合があります。

初期設定ファイル

今まで紹介した、エイリアスや環境変数はシェルを終了またはログアウトすると失われてしまいます。シェルには設定ファイルが福数個あり、その中に必要な設定内容を記述しておけば、いつでも好みの環境を再現することができます。以下に主な初期設定ファイルを解説します。

表 4:bash にまつわる初期設定ファイル

ファイル名	解説
/etc/profile	システム全体に対して有効、ログイン後最初に実行される(B-shll 系共通)
/etc/bashrc	システム全体でシェルが起動する度に実行。
~/.bashrc	ユーザで設定可能。bash が起動されるたびに実行される
~/.bash_profile	ユーザで設定可能。bash でログインする度に事項される。
~/.logout	ユーザで設定可能。ログオフ時に実行される(B-shell 系共通)
~/.bash_logout	ユーザに設定可能。ログオフ時に実行される。

bash をシェルにしているユーザがログインすると、上記表の順に初期設定ファイルが評価されます。
/etc/profile ば B-shell 系で共通となっており、ksh, bsh も初期設定ファイルとして実行されます。
ログイン時に実行とは、Login: と表示されているコンソールからログインした時、sshなどリモートコンソールからログインした時を指します。
シェルの起動とは、su や chgrpを実行した時を指します。関連して suコマンドを何度も繰り返すと、シェルの上にシェルを実行する事になり、コンピュータリソースを浪費します。必要な処理がおったら、必ず終了するよう心がけてください。
C-shell 系では同様に/etc/csh.login, /etc/csh.profile, ~/.login, ~/.cshrc があります。

クオーティショニング

シェルはキーボードから入力された文字列の中から、特別な意味を持つ文字を見つけるとその意味を解釈しようとします。たとえば、空白やタブはコマンドと引数を区切るという意味があります。
他にもドルマーク($)は、続く文字列を変数として、その値と置き換えます。

- シングル・クオーテーション(')
  全ての特殊文字の機能を無効にする。

- ダブル・クオーテーション(")
  変数の評価($)は行う。

- バック・クオーテーション(`)
  囲んだ文字列をコマンドとして実行した結果と置き換える。

Linux BASIC/MASTER や問題集に登場する簡単な例は以下のようになります。

```
$ echo "Hello, $USER"
Hello, student
$ echo 'Hello, $USER'
Hello, $USER
$ echo "Your current directory `pwd`"
Your current directory /home/student
```

また少し変わった事例として、クオーティショニングは [Enter](改行)も無効化します。それを利用すれば複数行にわたる、改行を含む echo を使う事ができます。

```
$ echo "`date` report
>
> Users
> `w`
>
> Disks
> `df`
> "
Tue Jan 17 13:55:43 JST 2012 report

Users
 13:55:43 up  6:33,  2 users,  load average: 0.00, 0.00, 0.00
USER   TTY      FROM              LOGIN@   IDLE   JCPU   PCPU WHAT
student       tty1     -        08:22    0.00s  0.21s  0.00s /bin/bash /home
student       pts/0    -        13:54    0.00s  0.01s  0.01s bash -i

Disks
Filesystem        1K-blocks      Used Available Use% Mounted on
/dev/sda2          17981340   2622948  14430240  16% /
/dev/sda1            295561     16251    264050   6% /boot
tmpfs                517348         0    517348   0% /dev/shm
```

またバック・クオーテーションの使い方として、よくあるパターンとして「ファイル一覧が書かれた」ファイルから、その詳細を ls で得る場合があります。たとえば /etc/shells に書かれたファイルの詳細を見る場合には、次のようにすると便利です。

```
$ ls -l `cat /etc/shells`
-rwxr-xr-x 1 root root  735804 Jul 22 10:15 /bin/bash
lrwxrwxrwx 1 root root       4 Oct 21 23:04 /bin/csh -> tcsh
lrwxrwxrwx 1 root root      21 Oct 21 23:04 /bin/ksh -> /etc/alternatives/ksh
lrwxrwxrwx 1 root root       4 Oct 21 23:03 /bin/sh -> bash
-rwxr-xr-x 1 root root  346756 Nov  4  2010 /bin/tcsh
-rwxr-xr-x 1 root root  514672 Jan 21  2011 /bin/zsh
-rwxr-xr-x 1 root root    5220 Mar 10  2011 /sbin/nologin
```

計算

シェルで計算するには、expr(1)コマンドを用います。expr はコマンドなので、数値と演算子はそれぞれ別の引数として投入する必要があります。また乗算(*)は、「カレントディレクトリのファイル全て」という意味があるため、注意が必要です。

```
$ expr 5 / 2
2
$ expr 2 * 3          ←この時、*はファイル一覧と解釈されてしまう。
expr: syntax error
$ expr 2 ¥* 3         ←バックスラッシュ（日本語 KB は¥）で特殊文字打ち消し
```

expr はどのシェルでも使えますが、bash であれば $ ((~))を使って、もっと簡単に記述する事ができます。これは直接シェルが解釈するため、数値と演算子を「分ち書き」にはしません。

```
$ echo $((2*3))
6
$ echo $((10/3))
3
$ echo $((10/2.5))
bash: 10/2.5: syntax error: invalid arithmetic operator (error token is ".5")
```

expr、$((～))とも整数しか扱えません、もし実数や三角関数などを手軽に計算したいのであれば、bc(1)コマンドを利用します。対話形式で、数式を入力すると答えが表示され、quit で終了します。

```
$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
2*3
6
10/3
3
scale=10
10/3
3.3333333333
quit
```

変数 scale に値を設定すると、計算する小数点以下の桁数を調整できます。また起動時に –l オプションを指定すれば簡単な三角関数などを利用する事ができます。詳しくはマニュアルを参照してください。

```
$ bc -l
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
s(1)
.84147098480789650665
scale=100
s(1)
.8414709848078965066525023216302989996225630607983710656727517099919¥
10404391239668948639743543052695 8
obase=16
127
7F
obase=2
10
1010
quit
```

さらに変わったところでは、units(1)があります。対話型に変換したい数値と単位、次に変換したい単位を入力すると相互変換結果を表示します。変換テーブルを使っているため(/usr/share/units.dat)日々変動する為替レートなどは、あまり意味がありません。

```
$ units
2438 units, 71 prefixes, 32 nonlinear units

You have: 2inch
You want: cm
      * 5.08
      / 0.19685039
You have: 9degC
You want: degF
      * 16.2
      / 0.061728395
You have: 20USD
You want: JPY
      * 2117.114
      / 0.00047234112
You have: ^D
$
$ wc /usr/share/units.dat
  4661  22576 204061 /usr/share/units.dat
```

コマンド処理（ファイル処理）

シェルの使い方を解説しましたが、さらにシステム管理や自動化で役立つコマンドを紹介します。シェルでプログラムを作るまえに、基本的な部分を抑えておく事は重要です。

ファイルの比較

システム管理と開発の両方で日常茶的に行われることは、ファイルの比較でしょう。システム管理なら設定ファイルやログ、開発ならソースコードの差分をチェックする事はよくあります。

システム管理でよく使う設定ファイルが変更されているか否か、変更されているのであれば、どこが変わっているのかを調べる場合を考えてみます。Apache HTTPD の定義ファイル httpd.conf と、インストール直後に httpd.conf.org として保存しておいたファイルを比較する場合、まずは ls(1)で確認するのが普通ですが、残念ながらファイルの大きさは同じ、更新日付が違うだけです。もしかすると単にファイルをコピーしただけかもしれません。

```
$ ls -l httpd.conf*
-rw-r--r-- 1 root root 33726 Jan 17 18:47 httpd.conf
-rw-r--r-- 1 root root 33726 Jan 17 18:42 httpd.conf.org
$ wc httpd.conf*
  991  4834 33726 httpd.conf
  991  4834 33726 httpd.conf.org
 1982  9668 67452 total
```

念のため wc(1) で比較しても、行数、単語数、文字数とも同じです。こんな時には diff(1)を使ってどこが違うかを検出します。

```
$ diff httpd.conf*
251c251
< ServerAdmin webmaster@myhome.ne.jp
---
> ServerAdmin root@localhost
281c281
< DocumentRoot "/www/html"
---
> DocumentRoot "/var/www/html"
306c306
< <Directory "/www/html">
---
> <Directory "/var/www/html">
$
```

この例では、比較したファイルの 251 行目、281 行目、306 行目がそれぞれ変更(c:change)されていて、その内容を表示しています。この例では行数は変わらず、内容の変更だけでしたが、実際にはもう少し複雑です。

```
$ cat data1.txt
# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1     localhost.localdomain localhost
::1           localhost6.localdomain6 localhost6
192.168.182.128 h128.s182.la.net h128

$ cat data2.txt
# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1     localhost.localdomain localhost myhost
192.168.182.128 h128.s182.la.net h128
192.168.182.1 h001.s182.la.net gw182.la.net
```

```
$ diff data?.txt
3,4c3
< 127.0.0.1   localhost.localdomain localhost
< ::1         localhost6.localdomain6 localhost6
---
> 127.0.0.1   localhost.localdomain localhost myhost
5a5
> 192.168.182.1 h001.s182.la.net gw182.la.net
```

この例では、data1.txt、data2.txt を比較しています。data1.txt の 3,4 行目の 2 行が、data2.txt の 3 行目になっている(変更されたのは 3 行目で、data1.txt の 4 行目は削除されている)。data1 の 5 行目の後ろに 1 行追加し、data2 の 5 行目になっている事を表しています。
a や c の左辺が diff の引数で指定した左側のファイル(data1)における行数、右辺が同様に右側のファイルを表しています。
慣れないと分かりづらいので、そんな時は –c オプション(context、内容表示)を用いるとよいでしょう。

```
$ diff -c data?.txt
*** data1.txt 2012-01-17 19:00:22.000000000 +0900
--- data2.txt 2012-01-17 18:58:43.000000000 +0900
***************
*** 1,5 ****
# Do not remove the following line, or various programs
# that require network functionality will fail.
! 127.0.0.1   localhost.localdomain localhost
! ::1         localhost6.localdomain6 localhost6
  192.168.182.128     h128.s182.la.net h128
--- 1,5 ----
# Do not remove the following line, or various programs
# that require network functionality will fail.
! 127.0.0.1   localhost.localdomain localhost myhost
  192.168.182.128     h128.s182.la.net h128
+ 192.168.182.1 h001.s182.la.net gw182.la.net
$
```

また少し変わった比較として comm(1)があります、2 つのファイルを比較し、共通項をインデント(タブによる段づけ)により表します。インデントなし、一番先頭から行が始まるものは左辺のファイルのみに存在する行、インデント1つは右辺のファイルにのみあるもの、インデント 2 つは両方に共通する行として仕訳ます。オプションで左辺を 1、右辺を 2、共通を 3 として不要な出力を抑制できます。comm -12 とすれば両者に共通の部分だけを抜きだす事ができます。

```
$ comm data*.txt
              # Do not remove the following line, or various programs
              # that require network functionality will fail.
127.0.0.1     localhost.localdomain localhost
      127.0.0.1       localhost.localdomain localhost myhost
      192.168.182.128 h128.s182.la.net h128
      192.168.182.1   h001.s182.la.net gw182.la.net
::1           localhost6.localdomain6 localhost6
192.168.182.128 h128.s182.la.net h128

$ comm -12 data*.txt
# Do not remove the following line, or various programs
# that require network functionality will fail.
```

この様にファイルの大きさが同じであっても内容が異なる場合あがります。ファイルサイズだけでは真贋を見抜けないので、インターネット上で公開されているファイルはチェックサムという特殊な計算によってファイル内を検査した結果を公開しています。

```
$ md5sum httpd.conf*
470d10809b945d28beb7798b7de14897  httpd.conf
cb48219f55c9a08a1c3aad20b0a89561  httpd.conf.org
```

```
$ sha256sum httpd.conf*
282a02dc2d2c5763027141201d8ff92a2fb098a042a5f4dd167dd7977281314b  httpd.conf
1db93f9d70436234707cf301c3974166756a5fed0f11f9a82f53ba05e747aa86
httpd.conf.org
```

ファイルを探す

ファイルを探すには、find(1)を使いますが、単純なファイル名検索の場合は locate(1)が有効です。find が実際にハードディスクを走査しファイルを探しますが、locate は事前に作成したデータベース情報にアクセスするため、格段に処理が早くなります。欠点は updatedb(8)によりファイル情報を更新しない限り、新しく作成したファイルは検出できない事です。
time(1)を使って、実測した例を示します。locate が約 20 倍高速である事がわかります。

```
$ time find / -name "*bash.ps" 2>/dev/null
/usr/share/doc/bash-3.2/rbash.ps
/usr/share/doc/bash-3.2/bash.ps

real  0m0.461s        ← 全体を通してかかった時間
user  0m0.174s        ← アプリが消費した時間
sys   0m0.271s        ← OS が消費した時間

$ time locate bash.ps
/usr/share/doc/bash-3.2/bash.ps
/usr/share/doc/bash-3.2/rbash.ps

real  0m0.023s
user  0m0.021s
sys   0m0.002s
```

find は様々なオプションがありますが、特にファイル属性（パーミッション、ファイル種類、タイムスタンプ）を使った検索には効果的です。よく使うオプションは以下の通りです。

- -type ファイル種別
  ファイル種別を 1 文字の記号で指定します。(f:ファイル、d:ディレクトリ、l:シンボリックリリンク)
- -name “ファイル名”
  ファイル名（ディレクトリ）名を指定します。ワイルドカードも使えます。
- -maxdepth　ディレクトリ深さ
  検索対象を指定したディレクトリ階層までに絞ります。たとえば指定したディレクトリより下を検索しない場合は 1 を、1 つ下のディレクトリまでなら 2 を指定します。
- -mtime/ -atime +/- 日付
  最新更新日付(mtime)またはアクセス日付(atime)を指定します。－は指定した過去日から今日までを、+は指定した日から過去を表します。

	本日				
日付	18	17	16	15	14
−1					
−2					
+2					
+1					

−mtime / −atime

  似たオプションに「–newer ファイル」があり、指定したファイルより新しいものを探します。
- –perm –パーミッション
  指定したパーミッションを持つものを検索します。-perm -002 又は –o+w なら第三者が書込み可能を意味します。マニュアルに沢山のサンプルがあるので詳しくはそちらを参照してください。

```
$ find /bin /usr/bin -perm +6000
```

  SetUID または GID が付与されているファイル一覧

find は検索した結果、そのファイルに対して操作を行う事ができます。-exec または-ok オプションで、後者は実行時に確認を行います。

find 開始ディレクトリ 種々の検索条件 –exec 操作コマンド ¥;

この時操作コマンドの終わりを示すのがセミコロン(;)ですが、シェルの区切り文字ではないため、打ち消しのバックスラッシュが必要になります。
操作コマンド内で検出したファイル名を参照するには、これもシェルに解釈されないよう‘{}’ を用います。

・大きなファイルサイズ(2MB 以上)コマンドの一覧

```
$ find /bin /usr/bin -size +2M -exec ls -ld '{}'  ¥;
-rwxr-xr-x 1 root root 4623872 Jan  6  2007 /usr/bin/doxygen
-rwxr-xr-x 1 root root 3828168 Apr 21  2011 /usr/bin/Xnest
-rwxr-xr-x 1 root root 3886596 Jul 22 11:36 /usr/bin/gdb
-rwxr-xr-x 1 root root 4380816 Apr 21  2011 /usr/bin/Xvnc
-rwxr-xr-x 1 root root 4144360 Apr 21  2011 /usr/bin/Xvfb
-rwxr-xr-x 1 root root 3778560 Feb  1  2011 /usr/bin/crash
```

・一年以上変更されていないファイルを確認しながら削除

```
$ find . -type f -mtime +365 -exec ls -ld '{}' ¥; -ok rm '{}' ¥
-rw-rw-r-- 1 student student 0 Jan  2  2011 ./lastyear
< rm ... ./lastyear > ? y
-rw-rw-r-- 1 student student 0 Jul  5  1966 ./longtimeago.txt
< rm ... ./longtimeago.txt > ? n
```

ファイルを調べる

使途が分からないファイルや、何をしているプログラムなのかを判別するためには、そのファイルの属性を調べるだけでは不十分です。このファイルは何かを調べる方法としては次のようなものがあります。

- どのパッケージに含まれているのか？
  rpm(8)コマンドを使って、このファイルはどのパッケージに含まれているかを探す事ができます。

```
$ rpm -qf /usr/sbin/suexec
httpd-2.2.3-53.el5.centos
```

- どんな種類のファイルなのか？
  file(1)を使えば、ファイルの種別と属性情報を表示することができます。

```
$ file /bin/bash /etc/init.d/httpd /var/www/icons/left.*
/bin/bash:           ELF 32-bit LSB executable, Intel 80386, version 1 〜
/etc/init.d/httpd:   Bourne-Again shell script text executable
/var/www/icons/left.gif: GIF image data, version 89a, 20 x 22
/var/www/icons/left.png: PNG image data, 20 x 22, 1-bit colormap, non-interlaced
```

- どんな機能（ライブラリ）を使っているのか？
  ldd(1)を使ってプログラムが呼び出すライブラリをチェックすることで、ある程度何をするプログラムなのか同定することができます。たとえば libwrap なら TPC ラッパーを使っているとか、libcrypt であればログインパスワードを使っている等がわかります。

```
$ ldd /bin/cat
    linux-gate.so.1 =>  (0x0070d000)
    libc.so.6 => /lib/libc.so.6 (0x00400000)
    /lib/ld-linux.so.2 (0x003dc000)
```

・ とにかく手がかりが欲しい
strings(1)で表示可能な文字列をとにかく取り出し、コメント情報やヘルプ、固定値から内容を調べるという方法もあります。古いシステム(1990 年代頃まで)では @(#)に続けてバージョン情報が記述されていましたが、最近は少なくなりました(シェルに多い)。

```
$ strings /bin/ls
/lib/ld-linux.so.2
Nj64
PTRh        ←  意味不明な文字列も表示される
    :(中略)
[^_]
Try `%s --help' for more information.
Usage: %s [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort.
```

コマンド処理（フィルタ）

UNIX/Linux は CUI なので、多くのコマンドを覚える必要があります。常に manpage や Google が必要になるので、敬遠する人も多いのは事実です。しかしコマンドを覚えるだけでは全く UNIX/Linux 使いにはなれません。そもそもコマンド（マニュアル第1章）は、CentOS5.7 で 1,200 以上も存在し、覚えきれません。

UNIX/Linuxを使いこなすには、種々のコマンドを組み合わせ、目的の結果をえるまで絞り込む操作が不可欠です。このようにコマンドの実行結果を、別のコマンドでさらに篩い落とす操作をフィルタと呼びます。

多くの機能を取り込み重厚長大になった MULTICS の反省から生まれた UNIX は、Simple Is Best を信条とし、「同じ発明をしない、すでにあるものを活用する」がコツです。

そのため UNIX/Linux のコマンドの殆どは味気ないメッセージしか出しません。これは後段で加工する際にはとても重要な事です。

この章では、UNIX/Linux を使う上での重要なコツ「フィルタ」を解説します。

ユーザの一覧

日本の企業では定期採用、いわゆる新卒採用を毎年同時期に行っています。つまり年に 1、2 回大量にユーザが増える事になります。こんな時、ユーザの一覧を作成する必要がありますが、コマンドの組み合わせで簡単に作成することができます。

cut(1)による項目の抽出

cut は指定した区切り文字でテキストファイルを区切り、必要な項目だけを抽出する事ができます。

cut　-d 区切文字　-f 位置

区切り文字は任意の半角 1 文字、位置は区切られた項目を左から 1, 2, 3 と連番にして指定します。間にハイフンを入れて、n-m のように n 番目から、m 番目という指定もできます。

今回は、passwd(5)から、ユーザ名、UID、GECOS 領域を抽出する事とします。

例）cut を使って必要な項目の抽出

```
$ cut -d: -f1,3,5 /etc/passwd
root:0:root
                （中略）
gdm:42:
oprofile:16:Special user account to be used by OProfile
student:500:YAKOSHI Akihito,Hakozaki,4411,xxxx
ycos:501:NARITA Brian,Ritto,9153
postfix:89:
tsukahara:502:TSUKAHARA Ichiro,Toranomon,7227,xxxx
nakamura:503:NAKAMURA Tohru,Shinbashi,8665,xxxx
shida:504:SHIDA Masao,Mishuku,7878
kimura:505:KIMURA Shouji,Otemachi,9926,xxxx
samejima:506:SAMEJIME Yuji,Shibuya,1618
yamato:507:YAMATO Mitsuru,Ichibancho,336
kusano:508:KUSANO Hideki,Ginza,2545
kawanishi:509:KAWANISHI Ayano,Ginza,2545
iwamoto:510:IWAMOTO Azumi,Shinbashi,5588
```

ソート（並び替え）

上記の結果を見てみると、postfix(UID=89)といった一般ユーザとは関係ない ID が混じっています。これはユーザの作成順に passwd ファイルへ追加されているためです。ユーザ ID 順に並び替えるには sort(1)を使います。

sort　-t 区切文字　-k 比較位置　[-n 数値として][-r 降順]

先の cut コマンドの後ろにパイプラインで sort を接合します。今回もコロン(:)を区切り文字にしています

が、抜き出したデータは<ユーザ名>、<UID>、<GECOS 領域>なので並び替える対象 UID は 2 番目となります。

例）先の例に、UID によるソートを追加

```
$ cut -d: -f1,3,5 /etc/passwd | sort -t: -k2 -n
       （省略）
student:500:YAKOSHI Akihito,Hakozaki,4411,xxxx
ycos:501:NARITA Brian,Ritto,9153
tsukahara:502:TSUKAHARA Ichiro,Toranomon,7227,xxxx
nakamura:503:NAKAMURA Tohru,Shinbashi,8665,xxxx
shida:504:SHIDA Masao,Mishuku,7878
kimura:505:KIMURA Shouji,Otemachi,9926,xxxx
samejima:506:SAMEJIME Yuji,Shibuya,1618
yamato:507:YAMATO Mitsuru,Ichibancho,336
kusano:508:KUSANO Hideki,Ginza,2545
kawanishi:509:KAWANISHI Ayano,Ginza,2545
iwamoto:510:IWAMOTO Azumi,Shinbashi,5588
nfsnobody:65534:Anonymous NFS User
```

今度は nsnobody が出現しました。この様に並び替えるだけでも新しい発見があり、ツールを使う重要性が分かると思います。

フィルタ中での編集

またコロンで区切った状態では見づらいので、区切り記号をタブに変換します。tr(1)で変換してもよいですが、sed(1)の方が応用範囲が広いので、今回は sed を用います。

sed -e ‘編集式’ [ファイル]		
主な編集式		
	s/文字列1/文字列 2/g	文字列置換。g オプションは 1 行全てを置換、ない時は最初に合致したものだけ置換。
	/文字列/コマンド	文字列にマッチした行に対して操作
	n コマンド	n 行目に対し操作
コマンド		
	d	削除
	a	改行に続く文字列を挿入

例）一般ユーザの一覧表を作成する

```
$ cut -d: -f1,3,5 /etc/passwd | sort -t: -k2 -n | sed -e '1i ¥
> ============== User List ===============' -e '/:5[0-9][0-9]:/s/:/¥t/gp -e d
============== User List ===============
student 500   YAKOSHI Akihito,Hakozaki,4411,xxxx
ycos  501     NARITA Brian,Ritto,9153
tsukahara     502     TSUKAHARA Ichiro,Toranomon,7227,xxxx
nakamura      503     NAKAMURA Tohru,Shinbashi,8665,xxxx
shida 504     SHIDA Masao,Mishuku,7878
kimura        505     KIMURA Shouji,Otemachi,9926,xxxx
samejima      506     SAMEJIME Yuji,Shibuya,1618
yamato        507     YAMATO Mitsuru,Ichibancho,336
kusano        508     KUSANO Hideki,Ginza,2545
kawanishi     509     KAWANISHI Ayano,Ginza,2545
iwamoto 510   IWAMOTO Azumi,Shinbashi,5588
```

インストール済みパッケージ一覧

パッケージの種類は非常に多く、CentOS 5.7 の配布 DVD では 2,700 を超えます。普通にインストールしても 1,000 前後と非常に情報量が多く管理が大変です。そこで、パッケージ一覧をいくつかの視点で作ってみましょう。

文字列の長さ

パッケージ名は非常に長いですが、実際に何文字ぐらいあるのでしょう。ファイル全体の行数や単語数は wc で算出できますが、ファイル名となると事情が異なります。
シェルの変数に入力さえできれば、${#name}展開で文字列を得る事ができます。

```
$ echo $USER
student
$ echo $USER | wc
      1       1       8
$ echo ${#USER}
7
```

(ちなみに、wc は行末の改行コードも数えてしまうため最後の単語は文字数+1 になります)

インストールされているパッケージは rpm で参照する事が出来ます、この結果を 1 行 1 行変数に代入すれば、パッケージ名の長さを求める事ができるはずです。
必要な要件としては
1. 標準入力(パイプラインの前段)から、逐一パッケージ名を読み取り変数へ代入する
2. パッケージの一覧全てを読み終えるまで、上記を繰り返す
という事になります。

1 行づつ読み取る

標準入力からデータを読み取り、変数へ代入するには read(BASH_BUILTINS(1))を使います。
read は入力があれば正常終了、何も入力がない場合は 1 を返します。

read 変数名	標準入力から文字列を入力し変数へセット

```
$ read ans
hoge
$ echo $ans $?
hoge 0                  ← 入力した文字と、成功(0)
$ read ans
^D                      ← 何も入力しない(^D)と、失敗(1)
$ echo $ans $?
1
```

繰り返し実行(while)

前段の処理が終わるまで繰り返すには while(BASH_BUILTINS(1))を使います。

while 条件 do 　処理… done	条件が真の間、do～done の間を繰り返す。 (条件にコマンドを指定すると、その結果が成功の間繰り返し)

先の read と組み合わせて次のようにすると、パッケージ名の長さと、パッケージ名を一覧表示します。

```
$ rpm -qa | while read pkg
> do
> echo ${#pkg} $pkg
> done
```

```
18 tzdata-2011g-1.el5
16 rmt-0.4b41-5.el5
       (以下省略)
```

この様に複雑なコマンドで間違った時は、fc(BASH_BUILTINS(1))を使って vi(1)で修正できます。保存すると、その内容が実行されます。また引数を指定すると、その文字列で始まるコマンド履歴を編集します。

さて、目的はパッケージ名の最大長さを測定するのですから、sort を使って文字数を並べ替えれば最後に答えが表示されます。

```
$ rpm -qa | while read pkg
> do
> echo ${#KpkgK} $pkg
> done | sort -n
10 bc-1.06-21
       :
43 xorg-x11-fonts-ISO8859-1-100dpi-7.1-2.1.el5
46 system-config-securitylevel-tui-1.6.29.1-6.el5
```

この環境で最も長いインストール済みパッケージ名は system-config-securitylevel-tui で 46 文字です。またインストール済みパッケージは 727 に上ります。
これだけ長く、量が多いとどういったパッケージがあるのか俯瞰するのは難しいと思われます。そこで、パッケージ名の一部を集約し把握しやすいように一覧を作りかえることにします。
パッケージ名はハイフンで区切られ、用途によって修飾されています。たとえば、

gnome-applets-2.16.0.1-19.el5
gnome-audio-2.0.0-3.1.1
gnome-backgrounds-2.15.92-1.fc6
gnome-desktop-2.16.0-1.el5.centos.1

などは GNOME のツール類です。そこでハイフンで区切った 1 つ目を代表名として集約して管理する事にします。上の例では gnome-xxxx は全て gnome グループとして管理するようなものです。

集約する

cut を応用すれば、以下のようなツールができます。しかし同じ名称が重複して表示されています。

```
$ rpm -qa | cut -d- -f1 |sort
       (省略)
yp
ypbind
yum
yum
yum
yum
zenity
zip
zlib
zsh
```

sort の –u オプションを使えば重複を集約してくれます。

```
$ rpm -qa | cut -Kd- -f1 |sort -u
       (省略)
yp
ypbind
yum
zenity
zip
zlib
zsh
```

これでかなり集約できたと思うのですが、まだ不十分です。たとえば lib で始まるパッケージや、同じような名前で末尾の数字（バージョン）が違うものなどが散見されます。これも sed を使って集約するとよいでしょう。

```
$ rpm -qa |cut -d- -f1 |sed -e 's/^lib.*/libXXX/' -e '/[0-9][0-9]*$/d' |sort –u
acl
acpid
alacarte
alsa
amtu
      :
yum
zenity
zip
zlibXXX
zsh
```

sed の補足）

・s/^lib.*/libXXX/　lib で始まるものは libXXX とする。

・/[0-9][0-9]*$/d　数字で終了するもの（[0-9]*は数字が 0 個以上の意味）を削除。

さらに集約した度数が知りたい場合には、uniq(1)の–c オプション（計数）を使い、以下のようにします。uniq 自身はソートを行わないので、前段でソートを行う必要があります。

```
$ rpm -qa |cut -d- -f1 |sed -e 's/lib.*/libXXX/' -e '/[0-9][0-9]*$/d' |
  sort | uniq –c
      1 acl
      1 acpid
      1 alacarte
      2 alsa
      1 amtu
    :
      4 yum
      1 zenity
      1 zip
      1 zlibXXX
      1 zsh
```

ちなみにライブラリはかなりの数があります。

```
$ rpm -qa |cut -d- -f1 |sed -e 's/lib.*/libXXX/' -e '/[0-9][0-9]*$/d' |
sort | uniq –c |grep lib
     2 cracklib
     1 glib2
     4 glibc
   101 libXXX
     1 pwlib
     1 zlib
```

シェルスクリプト

前章でコマンドを組み合わせれば、大抵の事は実現可能だと、わかって頂けたかと思います。有用なコマンドはファイルに保存しておけば、何時でも使え、また仲間と共有できて便利です。シェルスクリプト（台本）はプログラミング言語としてだけでなく、このようなノウハウの形式知化という目的でも利用されています。

約束ごと

配置場所

シェルスクリプトは単にテキストファイルですから、手軽に作成する事が出来ます。といって各自が好き勝手に似たようなスクリプトをあちこちに書き散らかすと、ノウハウの蓄積ができないどころか、微妙に動作が異なったり、バグを内包したりと業務に支障を来す場合があります。チームで共有すべきスクリプトは置き場所を明確に決め、更新ルールを決める必要があります。
また個人で使うスクリプトは ~/bin などに配置し、PATH 変数にもそのディレクトリを追加しておくとよいでしょう。PATH にカレントディレクトリ(.)を追加するのは、セキュリティの観点から好ましくありません。
上記の設置場所を決めたら、初期設定ファイルに追記しておくとよいでしょう。

~/.bashrc の内容：

```
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
      . /etc/bashrc
fi

# User specific aliases and functions
HISTSIZE=1000
HISTFILESIZE=1000
PATH=$PATH:~/bin
```

書き始め

最初の1行目は以下の形式となります。

#! （このスクリプトを実行させる）シェルのフルパス

まだシェルが B-shell しかなかった時代は問題なかったのですが、C-shell の登場により、どのシェルを使うかを判定する必要が出てきました。暫定的な処置として 1980 年代までは、先頭の 1 行目が#の場合は C-shell、何もないかコロン（: 、何もしないとうコマンド）であれば B-shell としていました。その後、ksh やシェル以外の Perl, awk といった言語が登場してくると、明確にスクリプトを実行するプログラム（スクリプトエンジン）を指定するようになりました。古い文献では、コロンで始まるスクリプトの記述があります。

コメントは豊富に入れる

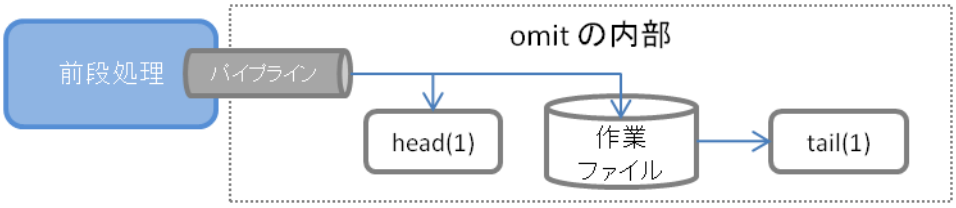
シェルスクリプトはシェルへの台本でもありますが、使い方や既知の問題、修正する場合の注意事項など利用する「人」に対する説明文でもあります。マメにコメントを記載するよう心がけてください。
以下は network の起動用スクリプトです。ほぼシェル向けの言語と人向けの説明文が1：1の比率で書かれています。

```
# Check that networking is up.
[ "${NETWORKING}" = "no" ] && exit 0

# if the ip configuration utility isn't around we can't function.
[ -x /sbin/ip ] || exit 1
```

基本パターン

では実際にスクリプト作成を行います。例題として、標準入力から得たデータの先頭部分と、末尾部分を表示する方法を考えます。マニュアルなどの実行例で使う事を想定し、名称は omit、引数は表示する行数で、ファイル処理は行わない(標準入出力のみ)ものとします。
標準入力から得た情報のうち、先頭部分を表示するには head(1)が、末尾部分は tail(1)が使えそうですが、入力は 1 系統しかありません。どこかに一旦データを保存する必要があります。



第 1 版(フィルタで試す)

まずは、コマンドラインで試すか、簡単なスクリプトを作り、実際に思った通りの動作かどうかを確認してみます。

```
#! /bin/bash

tee wk.dat | head -3 ; echo -e "¥t:" ; tail -3  wk.dat
rm -f wk.dat
```

解説)
各コマンドについては、過去に説明済みですので割愛します。新たなオプションとして echo の –e オプションがあります。これは特殊文字を出力するためのもので、¥t をタブとして出力します。
また作業用ファイルとして wk.dat を作成していますが、処理が終わったら rm で削除しています。
非常に簡単ですが、とりあえず最低限の機能は満たしているようです。

```
$ rpm -qa | omit
tzdata-2011g-1.el5
rmt-0.4b41-5.el5
dump-0.4b41-5.el5
      :
libhugetlbfs-1.3-8.2.el5
gnome-menus-2.16.0-2.fc6
```

しかし、ちょっとしたことで不具合が発生する事が分かります。基本的には作業ファイルの扱いです。
まずカレントディレクトリに書き込み権がないとエラー終了します。次に^C などで中断すると、作業ファイルは削除されません。またファイル名が固定なので、同時にいくつも omit が起動されると、作業ファイルを潰しあって正しい答えが出ないでしょう。

```
$ rpm -qa | ~/omit
tee: wk.dat: Permission denied
tzdata-2011g-1.el5
rmt-0.4b41-5.el5
dump-0.4b41-5.el5
      :
tail: cannot open `wk.dat' for reading: No such file or directory
```

第 2 版(後片付け)

作業ファイルは慣習として/tmp に書く事になっています。作業用なので不特定多数の人が書込む事ができます。しかし同じファイル名では同時に 2 つ以上動くと、競合してしまうため、プロセス固有の名称を付ける必要があります。そんな時は特殊な変数$$(プロセス ID)を使うのが常套手段となっています。
また確実に削除を行いたいのであれば、trap を使って、シグナルを受けるとコマンドを実行するよう、最初に設定しておきます。

```
#! /bin/bash
TMP=/tmp/omit$$
trap '/bin/rm -f $TMP' TERM QUIT EXIT

tee $TMP | head -3
echo -e "¥t:"
tail -3 $TMP
```

第 3 版（オプション処理）

最後に表示する行数をコマンドラインから指定できるよう、引数の処理を追加します。${#*}が引数の数を表し、$1, $2,…は引数の１番目、２番目を意味します。今回は引数がなければ既定値(3)とし、あればその値とします。2 つ以上の引数はエラーとします。

引数の判定は case を使って「引数なし」「1つのとき」「左記以外」の 3 つの条件判定を行っています。

case	比較対照	in	case は esac までて一つの処理になります。
値 1)	処理 1	;;	比較対照と値が合致すれば、その処理を行います。
値 2)	処理 2	;;	処理の終端は ;; で表します。
*)	処理 3	;;	値*は上記の判定以外(default)を表します。
esac			

```
#! /bin/bash
# @(#) stream output omitting filter (head and tail)
# usage: $0 [n]
TMP=/tmp/omit$$
trap '/bin/rm -f $TMP' TERM QUIT EXIT

case ${#*} in                   ← 引数の数をチェック
0)    line=3;;                  ← 引数がなければ表示行数を 3 とします
1)    line=$1;;                 ← 指定されればその値を採用
*)    cat <<EOD                 ← 想定外はメッセージを表示し、、
$0 is stream output omitting filter.
 output cut with head(1) and tail(1)
usage: $0 [n]
n is number of ouptput lines (default 3)
EOD
      exit 1;;                  ← 戻り値'1'で終了。
esac

tee $TMP | head -$line
echo -e "¥t:"
tail -n $line $TMP
```

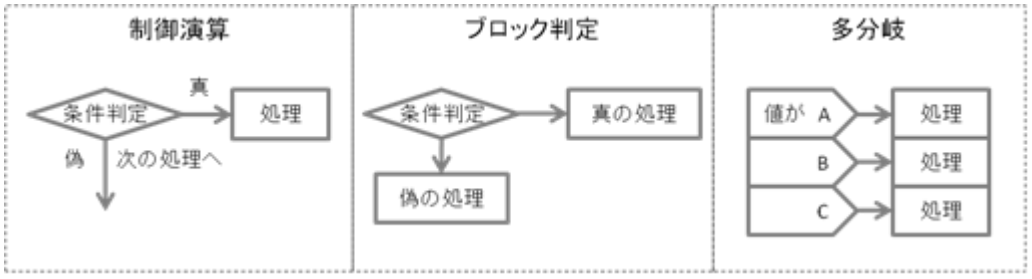
実際に引数を指定して動作を確認します。特にエラー処理は戻り値も確認しましょう。

```
$ omit 1 2 3
/home/student/bin/omit is stream output omitting filter.
output cut with head(1) and tail(1)
usage: /home/student/bin/omit [n]
n is number of ouptput lines (default 3)
$ echo $?
1
```

条件判断

コマンドの実行結果や、変数の値によって処理を実行すかしないかを決定するのが条件判断です。
bash では制御演算子（条件リスト）、ブロック条件、多分岐があります。
bash ではコマンドの戻り値が0の時「真」、0以外の時「偽」として扱います。ファイルの存在や数値の大小を判定するコマンドとして test 又は [～] があります。

図 2:条件判断のパターン



- 制御演算子
  直前の結果が真(0)か偽(0以外)によって、右辺を実行するかどうかを制御します。
  演算子は ＆＆または、｜｜で間に空白はありません。

  例)コマンドの実行結果により、その状況を表示する。

```
ping -c 1 host  &&  echo "network reachable"        ← 右辺が真の時、左辺を実行
[ -w / ]  ||  echo "Can't write a direcoty" ←  右辺が偽の時、左辺を実行
```

- ブロック判定
  bash では if、then, else, fi の組み合わせで制御を行います。複雑なパターンとして elfi もありますがここでは割愛します。

  例)UID によって root か、それ以外かを判断する。

```
if  [  －z  $UID ]
then
    echo "Root user"
else
    echo "Normal user"
fi
```

- 多分岐
  他の分岐と異なり真偽ではなく指定した変数や処理結果の値によって、処理を分岐します。
  case ～ esac までが多分岐の対象で、「値)」で指定し、;;までが一連の処理として扱われます。
  値に*を指定すると、何にでも合致し値に関係なく実施するという意味になります。
  判定は行の上から順に行われるため、*は最後に指定指定します。

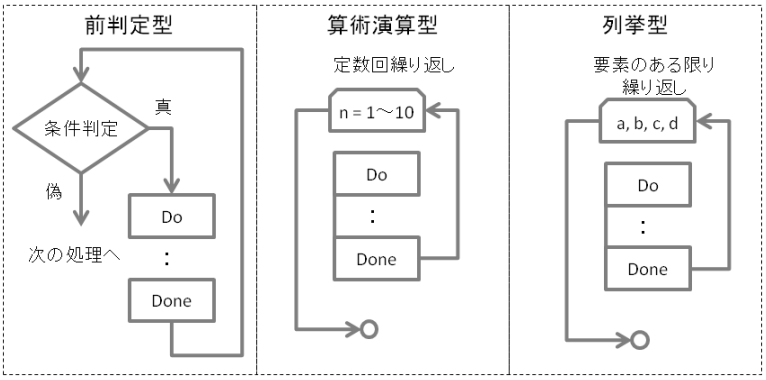
  例)変数 a の値によって表示を切り替える。

```
case $a in
0)  echo "Zero" ;;
1)  echo "One" ;;
2)  echo "Two" ;;
*)  echo "Many..."
    date ;;
esac
```

繰り返し処理（詳細）

シェルスクリプトを使う要件のうち、もっとも頻繁に必要とされるのが「繰り返し」です。bash の繰り返し処理には、前判定型、演算型、列挙型があります。
先のパッケージ名の集計で使った while は前判定型と呼ばれるものです。各型の概要は以下の通りです。

図 3:繰り返しのパターン



- 前判定型(while 条件式 do～done)
  まず条件式を評価し真であれば、do～done の間を実行、条件が真の間繰り返す。
  条件によって実行するかもしれないし、実行しないかもしれない。

  例)ファイル /tmp/loop.on がある限り、繰り返す
  ```
  while [ －f /tmp/loop.on ]
  do
      date
      sleep1
  done
  ```

  *「until 条件式」の場合は、真偽判定が逆となり、偽の間繰り返す。一般的なプログラム言語と異なり、bash に後判定はない。

- 演算型(for 演算式 do～done)
  指定された回数を実行する。演算式は 3 つの領域からなり、「初期値（開始）;終了条件（終了）;増分」となる。

  例)1～5まで繰り返す
  ```
  for ((n=1;n<=5;n=n+1))
  do
      echo $n
  done
  ```

- 列挙型(for 値並び do～done)
  並べた値について、順に実行する。

  例)カレントディレクトリの data*で始まるファイルを全て、～.bkup とう名称に変更する
  ```
  for f in data*
  do
      mv $f $f.bkup
  done
  ```

サンプル

スクリプトのサンプルと解説を掲載します。オリジナルは http://ycos.sakura.ne.jp/LA/ にセミナーと同じタイトルで用意しています(OpenLab20.tar.gz)。
なお LA のデフォルトユーザと同じ内容で、Basic 認証をかけています。

- ファイルを検索しマッチした部分を反転表示させる(shf)
  grep を使って文字列検索を行うと、ヒットした部分が分かりずらいので、反転させる例

shf

```
1 #!  /bin/bash
2 # @($) Search word and bold it.
3 # usage: $0 keyword [files..]
4 ptn=$1
5 shift
6 sed -e "s/$ptn/^[[7m&^[[27m/g" $*
```

4. 変数 ptr に第 1 引数を代入
5. 引数をずらす(旧引数 2→引数 1、旧引数 3→引数 2…)
6. 上記でずらした結果の引数 1～のファイル群について文字列変換
   ^[xxx はエスケープシーケンスといって端末の色や反転を制御する文字。
   ^[は[ESC]で、vi では入力時に^V を入力し[ESC]を入力

実行例)

```
[student@h128 bin]$ shf la.net /etc/hosts
# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1               localhost.localdomain localhost myhost
::1             localhost6.localdomain6 localhost6
192.168.182.128 h128.s182.la.net h128
[student@h128 bin]$ _
```

- 指定した回数コマンドを繰り返す(repeat)
  ベンチマークなど数回、同じコマンドを繰り返す場合などに利用。

repeat

```
 1 #! /bin/bash
 2 # command repeater, usage: REPEAT number command..
 3 num=$1
 4 shift
 5 while (($num>0))
 6 do
 7         let num=num-1
 8         eval "$*"
 9 done
```

3.～4. (shf)を参照のこと
5.while を使って、定数回繰り返す例
8. eval “文字列” は、文字列をコマンドとして実行

実行例)dd を使って 300MB のファイルを作成するベンチマーク

```
$ repeat 3 dd time dd if=/dev/zero of=dumy.dat bs=1M count=300
300+0 records in
300+0 records out
314572800 bytes (315 MB) copied, 0.642399 seconds, 490 MB/s

real    0m0.719s
user    0m0.000s
sys     0m0.546s
        :   (上記を繰り返す)
```

- コマンドの実行状況を保存(myscript)
  script を実行し、特殊文字(バックスペース等)を除去のうえ、MS-DOS フォーマットに変換

myscript

```
 1 #! /bin/bash
 2 # save script, backspace filter and
 3 format change
 4 DIR=/tmp
 5 rawf=$DIR/$1
 6 dosf=$rawf.txt
 7
 8 script -a $rawf
 9 cat $rawf | col -b > $dosf
10 unix2dos $dosf
```

  5～ 6. DOS に変換したファイルは引数名.txt としている。
  8.script(1)を追加書込みで実行
  9. 特殊文字フィルタ(col(1))の実行
  10.UNIX→DOS フォーマットに変換

- 同一ネットワークに存在するホストの一覧(allip)
  ネットワーク設定を読み込み、ブロードキャスト通信を行い返事のあった IP アドレスを一覧表示

allip

```
1 #! /bin/bash
2 # Get all active node IP address
3
4 bcast=`/sbin/ifconfig |sed -e '/Bcast/{s/^.*Bcast://;s/Mask.*$//p}' -n`
5 echo "Broadcast address ${bcast}"
6 ping -c2 -b $bcast >/dev/null 2>&1
7 /sbin/arp -a 2>/dev/null | sed -e 's/^.*(//' -e 's/).*$//' |sort -n
```

  4. ifconfig(8)の出力から、Bcas のある行を抽出し、Bcas:より前と、Mask から後ろを削除
  それ以外の結果は削除(ブロードキャストアドレスを抽出
  5. (4)の結果を表示
  6. ブロードキャストアドレスに ping を発信(全ホストへ問合せ)
  7. arp(8)を使い、回答のあったホストの一覧を表示

- 指定したファイル群の特定部分(行番の開始、終了位置を指定)取り出す(trunk)

trunk

```
 1 #! /bin/bash
 2 # a trunk of file / usage: $0 start-line end-line files...
 3 start=$1
 4 shift
 5 end=$1
 6 shift
 7
 8 for n in $*
 9 do
10         echo ">> $n <<"
11         sed -n -e "$start,${end}p" $n
12 done
```

  3~6. (shf)と同様
  8. 指定したファイル群を繰り返す(列挙型)
  9. sed を使い、指定行を取り出す。$[end]p は、変数 end と、sed のコマンド p(表示)を
  変数名 endp と誤解されないための処置

- ディレクトリを視覚的に表示 (dtree)
  指定したディレクトリをグラフで表示。sed による自作 tree –d

dtree

```
 1 #! /bin/bash
 2 # dtree ; Visual display of directory tree.
 3 dir=$1
 4 cd $dir
 5 pwd
 6 find $dir -type d -print | ¥
 7 sort -f | ¥
 8 sed -e "s,^$dir,," ¥
 9       -e '/^$/d'   ¥
10       -e 's,[^/]*/¥([^/]*¥)$,+----¥1,' ¥
11       -e 's,[^/]*/,|    ,g'
```

6. 指定したディレクトリ以下のディレクトリを表示
7. ソート(-f は英字の大文字小文字を区別しないオプション)
8. (6)の結果から、指定したディレクトリ部分を削除、末尾の¥は継続の意味
9. 空行(^は行頭、$は行末＝1 行に何もない行)を削除
10./～の文字列を 1 つの/に置換、最後のパターンを+---に置換
11./をインデント(空白)で置換

- 指定したファイルを監視(fbiff)
  ファイルを 1 秒ごとに関しし、警告音と共に作成された先頭部分を表示する。

fbiff

```
 1 #! /bin/bash
 2 # fbiff: If a file find then report it.
 3 while true
 4 do
 5         if [ -r $1 ]; then
 6                 echo "^G Found ! ^G"
 7                 echo " -------------- $1"
 8                 head $1
 9                 echo " --------------"
10                 exit
11         fi
12         sleep 1
13 done
```

3. while で無限ループ(処理の途中で終了、1/2 ループとも)
5. ファイルが読み取れるかどうかの判定
6. ^G はベル(Beep)、vi では^V を押し、続いて^G を押す
12. 1 秒間停止(これがないと、極端に CPU を消費するプログラムとなってしまう)

- システム概要の表示(myinfo)
  マルチベンダー環境でのシステム概要表示例。書式付編集の習作。

myinfo

```
 1 #! /bin/ksh
 2 # Start up login
 3
 4 echo ""
 5 printf "Host name: %-25s  CPU type: %s¥n" `hostname` `uname -m`
 6 printf "Terminal : %-25s  OS Type : %s %s¥n"  `tty` `uname -s` `uname -r`
 7 printf "Shell    : %-25s  LANG    : %s¥n" $SHELL $LANG
 8 printf "Current  : %s¥n" `pwd`
 9 date
10 uptime
11
```

5～8. printf(BASH_BUILDIN(1))による表示の編集例

- 索引見出しの作成(mkindex)
  標準入力から得たテキストデータをソートし、先頭 1 文字の爪見出しつき一覧を作成。

mkindex

```
1   #! /bin/bash
2   # Make Index for TEXT FILE
3
4   sed -e 's/[ ^I]*//' -e '/^$/d' $* |
5   sort -fd | awk '
6   BEGIN{
7           print "[a]"
8           old = "a" }
9   {
10          line=toupper($0)
11          key=substr(line,1,1)
12          if( old != key ) {
13                  old = key
14                  printf "[%s]¥n", old
15          }
16          printf "¥t%s¥n",$0
17  }'
```

4. 不要な文字を削除(空行、空白、タブ(^I))
5. 辞書順、大文字小文字区別なしでソート

- スケール(文字数メモリ)の表示(scale)
  指定した桁数のスケールを表示、5 の倍数は+、10 の倍数は 10 の桁を表示

scale

```
 1  #! /bin/bash
 2  # @(#) print column scale
 3  max=${1:-80}
 4  n=1
 5  m=1
 6  for ((n=1;n<=$max;n++))
 7  do
 8          case $n in
 9          *5)     echo -n "+" ;;
10          *0)     echo -n $m
11                  let m=m+1%10
12                  (($m>9)) && m=0
13                  ;;
14          *)      echo -n "." ;;
15          esac
16  done
17  echo ""
```

3. 引数1を変数 max に代入、省略値は 80(パラメータ展開の初期値)
6. 算術型の繰り返し
9. 繰返し数が 5 で終わる場合に+を表示、echo の –n オプションは改行なし
10.繰り返し数が 0 で終わる場合いは、繰返し数の 10 位(10 の除余)を表示

実行例)

```
$ scale 50
….+….1….+….2….+….3….+….4….+….5
```

付録内容

ファイル名	概要	関数	パラメータ置換	数値演算	ヒアドキュメント	shift, set	条件演算子	if-then	case	while	for(算術)	for(列挙)	getopts	sed	sort	awk	その他	行数
allip	活動ホスト一覧													○	○		ifconfig, ping	8
calc	簡易電卓		○	○	○	○	○			○			○				bc	46
cntr	センタリング		○			○		○	○							○		35
dice	サイコロ(乱数)			○				○	○									25
dtree	ディレクトリ構造表示													○	○		find	11
fbiff	ファイル作成監視							○		○							head	13
fstat	ファイル属性詳細																perl	67
gap	空き連番検出					○			○				○		○	○		38
htbl	HTML形式作表					○			○				○			○		34
mkindx	爪見出し索引作成													○	○	○		17
mn	数当てゲーム			○			○	○	○	○								8
myinfo	システム概要表示																uname, printf	10
myscript	操作記録																script, col, unix2dos	9
now	単語の計数				○			○	○				○			○	col, uniq	37
omit	ファイルの中略				○				○								trap, head, tail, tee	23
pseq	連番を伴う文字列生成					○			○							○		23
repeat	コマンドの繰り返し			○		○				○							eval	9
rn	一括ファイル名変換	○	○	○	○	○		○	○	○		○				○		107
scale	スケール表示		○	○					○		○							20
shf	目立つ文字列検索					○								○				6
total	一覧の合計を表示															○		19
trunk	連番の空きを表示				○	○						○		○				12
ydup	行の連結			○		○			○				○	○		○		42