

Scripting

— シェルスクリプト (bash) 前編 —



田口 景介

あなたは入力した文字を取り消すとき、Ctrl-HとBackspace、どちらを使っているだろうか。筆者はCtrlキー至上主義者なので、当然Ctrl-Hだ。それだけでなく、Enterの代わりにCtrl-J、Tabの代わりにCtrl-I、ESCの代わりにCtrl-[と、できるだけ特殊キーには触らず、徹底してCtrl+メインキーで操作している。なぜかと言えば、その方が速いからだ。特殊キーはキーボードの外周に配置されているので、いちいちホームポジションから手を離さなければならないし、ホームポジションに戻すのもわずらわしい(それなのになぜ「変換」キーを使うのかと人から言われるが、いいのだ。あれは近い)。別にこれが筆者の好みだというだけで、人に勧めるつもりはないが、ユーザーが習熟すればするほど効率よく、快適になる。それこそがLinux文化、ひいてはUNIX文化であろう。キー操作だってその方がいい。

当コーナーでとりあげる「スクリプト」もそんな文化の一端を担う、噛めば噛むほどに味が出る、おいしいテクニックである。一般的には、頻繁に実行する一連の手続きをファイルに記録したものがスクリプトと総称されている。規模の大きなアプリケーションに装備されているマクロ機能もスクリプトと同等の意味で使

われる言葉だ。一連の手続きと述べたが、たいていのスクリプトは制御構造(条件分岐やループ)を持ち、変数を扱えることから、順に処理をこなしていく単純なものだけではなく、プログラミング言語並みの処理だって記述できる。たとえば、カーネルのリコンフィグを行うとき“make menuconfig”とするが、このとき実行されるメニュー形式のプログラム(Menuconfig)は実はスクリプトである(画面1)。もっともこれほどの処理にあえてスクリプトを使うのは、かなり趣味的な意図を感じないでもない。普通は長くても数十行程度で記述できるような、小技的ツールの作成にスクリプトを利用す

ることのほうが多い。ただし、一般的にスクリプトはテキストファイルとして作成するので、CPUやOSのバージョンなどの動作環境にほとんど依存せずに実行可能だ。この点が重視される場合にはその限りではない。また、ユーザーが簡単に変更したり、動作を確認したりできるのもスクリプトのメリットの1つである。

ところでLinuxには、シェル、perl、ruby、python、tcl/tkなど、さまざまなスクリプトを実行する処理系が実装されている。これらの処理系はそれぞれに得意分野があり、また文法や機能が異なるので(重なる部分も多いが)1つ覚えておけば十分というものでもない。そこで、

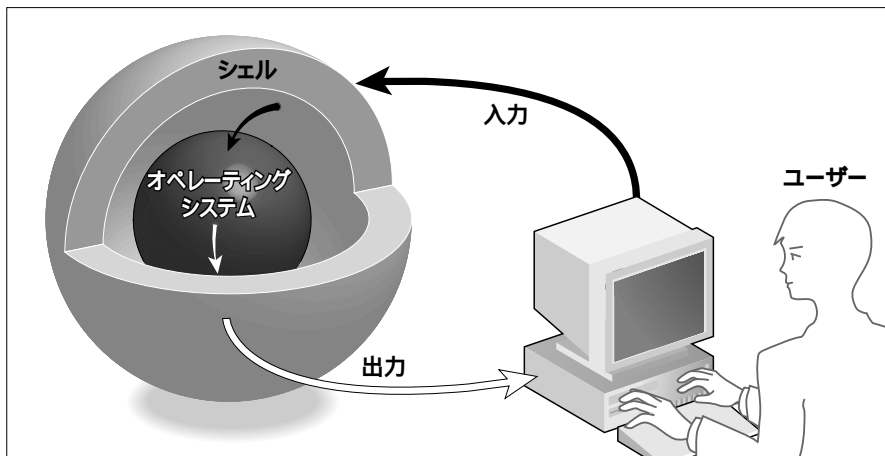


図1 シェルは、ユーザーとオペレーティングシステムとの仲介役

当コーナーではLinuxで利用できるスクリプト言語をいくつかピックアップし、解説していく予定である。雑誌というメディアの性格上、スクリプト言語の機能をすべて網羅するリファレンスのな解説はしない(必要ならば書籍やWWWを参照して欲しい)。覚え始めのとっかかりをつけやすくし、なおかつTipsを紹介しながら進められればと考えている。

システム管理には欠かせない シェルスクリプト

さて、今回は誰もが利用するスクリプトから始めようということで、シェルスクリプトをとりあげることにした。

Linuxユーザーならば、シェルスクリプトを作ったことがなくても、シェルスクリプトを実行したことがないなどありえないことだ。カーネルの再構成を行う程度の知識があれば、先のMenuconfigがシェルスクリプトであることぐらいは知っているかもしれない。しかし、多くのユーザーは/usr/bin/groupsがスクリプトだと知らずに利用しているのではないだろうか。zlessやzdiffなどのzシリーズはどうだろう。実際、意外と知らず知らずのうちにシェルスクリプトを使っているのである。

利用しているコマンドがシェルスクリプトかどうかは、fileコマンドで調べられる。たとえば“file /usr/bin/groups”とすれば、“/usr/bin/groups: Bourne shell script text”と表示され、シェルスクリプトであることがわかる。また、たいいのシェルスクリプトはC言語などで開発されたコマンドに比べてファイルサイズが小さくなる傾向があるので、次のコマンドを実行して先頭付近に表示されるコマンドのうち、シンボリックリンクを除いたほとんどのファイルはシェルスクリプトと判断してもいいだろう。

```
bash$ ls -l /usr/bin | sort -n +4
```

このように、シェルスクリプトと一般的なコマンドは一見ただけでは区別できないし、する必要もない。後述するよ

うに、スクリプトファイルに適切な設定を施せば、シェルスクリプトに限らず、スクリプトは一般的に通常のコマンドと同等の扱いが可能だ。ただし、あえてシェルスクリプトであることを主張するために、ファイルの拡張子として.shをつける場合もある。

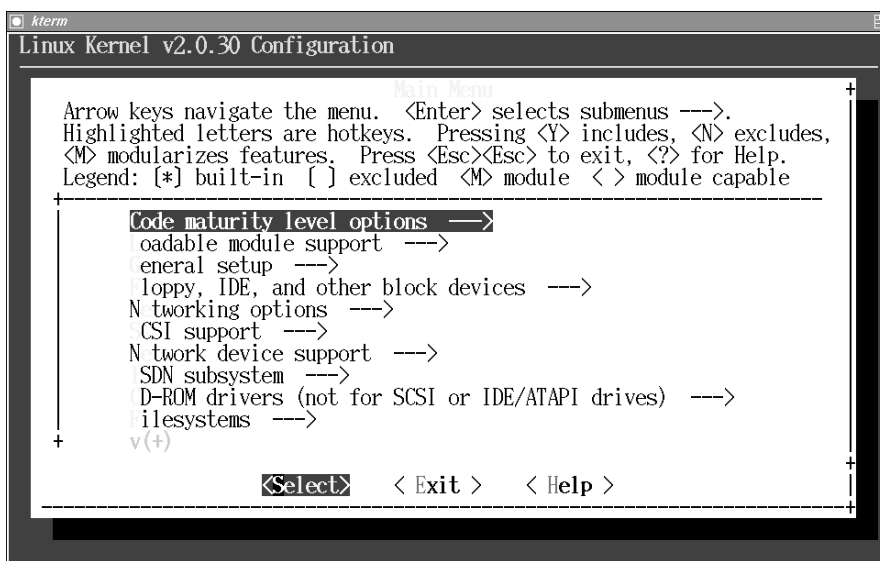
また、ユーザーのホームディレクトリに格納する.bash_profile、.bashrc、.xinitrc、.xsessionなどのスタートアップスクリプトもシェルスクリプトである。これらのファイルはユーザーが明示的に実行するわけではないが、ログインすれば必ず実行される。最もユーザーに身近なシェルスクリプトと言えるだろう。今回はこの.bash_profileと.bashrcを参考に、シェルスクリプトの作り方を解説する。

さらに、/etc/rc.dなど(ディストリビューションによって異なる)に格納されている、Linuxのブートシーケンスが記述されているファイルもシェルスクリプトである。ブート時には、/etc/inittab、/etc/rc.d/rcから始まり、シェルスクリプトがシェルスクリプトを繰り返し呼び出し、/etcフォルダに格納された十数個のシェルスクリプトが実行され、最後にやっとlogin:プロンプトが表示されるのだ。つまり、Linuxの環境構築を行うた

めには、シェルスクリプトの理解が欠かせないというわけだ。管理者たるものシェルスクリプト程度理解できて当然、とまではいわないが、最低でもシェルスクリプトを読んで理解できるだけの知識は持ち合わせておくべきだろう。そうすれば、インストーラで設定したIPアドレスを変更するにはどこを修正すればいいのか、なんてことで悩む必要もなくなる(ちなみにRedHat 5.2の場合、IPアドレスは/etc/sysconfig/network-scripts/ifcfg-eth0に記録されている)。

Linuxの標準シェルはbash

テキストエディタにemacsやviがあるように、シェルもアプリケーションのジャンルの名前であり、いくつものシェルが作られている。つまるところ、シェルスクリプトはシェルによって実行されるからシェルスクリプトと呼ばれるわけだが、シェルスクリプトと呼ばれるスクリプト言語があるわけではない。シェルによって機能や文法は異なっている。もっとも、基本的には、sh / ash / bashなどが属するsh系と、csh / tcshなどが属するcsh系の二系統に大まかに分類できるので、どちらかを選ぶことになる。歴史的に古く、



画面1 Menuconfigスクリプト

Linuxのカーネルコンフィグレーションを行うときに利用するMenuconfigはシェルスクリプトで記述されている。ウィンドウシステムライクな画面の描画には専用のプログラムをスクリプトから呼び出して利用しているが、最小限のハードウェア依存コードで動作するメリットは大きい。

システム関連スクリプトで使われることが多いsh系スクリプトと、C言語ライクな文法を取り入れたcsh系スクリプト。機能的にどちらが勝っているということではなく、処理速度が大きく違うわけでもない。結局はユーザーの好みで選ぶしかない。ただ今回は、sh系に属するbashをとりあげることにする。その理由は、Linuxではユーザーアカウント作成時のデフォルト設定がbashであること（ディストリビューションによって違うものがあるかもしれないが）、そして/etc以下に格納されているシステム管理用シェルスクリプトのほぼすべてがshのスクリプトであることだ。このためLinuxユーザーは必然的にbashと付き合う機会が多くなるはずで、csh党のユーザーがあえてcshに切り替えたりしない限りは暗黙のうちにbashを利用することになるだろう。それならば、bashのシェルスクリプトを修得しておいた方が得というものだ。

ところで、ユーザーが使用しているシェルの種類を知りたいければ、次のコマンドを実行すればよい（bash\$はプロンプトを表す）。

```
bash$ grep ^"$USER" /etc/passwd | cut -d: -f7
```

すると、“/bin/bash”のように管理者あるいはユーザーが設定したシェルが表示される。もしそれが“/bin/bash”ではなく、管理者が設定したものであれば、管理者はそのシェルをいたくお気に入りということだ。もしbashに変更して欲しいと頼むのなら、少し控えめな表現しておいた方がいいかもしれない。間違っても「なんでxxxになっているんですか？」などとは言わないことだ。下手をすると長話に付き合わされる羽目になるかもしれない。

また/usr/bin/chshコマンドを使って、ユーザー自身が自分のシェルを変更することもできる。ただし、/etc/shellsにリストアップされていないシェルに切り替えることはできないので、目的のシェルがリストになれば、やはり管理者にお

願いする必要がある。



1行スクリプト

前置きはこれぐらいにして、シェルスクリプトに話題を移そう。まずはごく基本的な1行のスクリプトを作ることから始める。

テキストエディタを使って、次の1行からなるファイルを作り、showshという名前をつけて保存して欲しい。これは先に紹介した、ユーザーのログインシェルを表示するコマンドである。このように、シェルのコマンドラインに入力する内容をファイルに保存すれば、それだけでそのファイルはシェルスクリプトファイルと呼べるのだ。

```
grep ^"$USER" /etc/passwd | cut -d: -f7
```

このshowshスクリプトを実行するには、

```
bash$ . showsh
```

あるいは

```
bash$ source showsh
```

と入力する。どちらもshowshファイルを読み込み、キーボードから入力された場合と同じように実行される。2種類方法が用意されているのは、sh系のコマンドであるドット（.）とcsh系のコマンドであるsourceコマンドの両方をbashが取り込んでいるからだ。このようにsh系のコマンドは記号的な名前、csh系のコマンドは読みやすくコマンド名から機能が想像しやすい名前が多く使用されている。

スクリプトを実行するには、もう1つ方法がある。これには、先のshowshファイルの先頭行に次の1行を追加する（「#!」の前には空白を開けず、行頭から入力すること）。

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

さらに、次のコマンドを実行し、showshファイルに実行属性（x）を付ける。

```
bash$ chmod +x showsh
bash$ ls -l showsh
-rwxrwxr-x 1 keisuk-t keisuk-t
53 Feb 13 07:24 showsh
```

以上2つの設定を行うと、次のようにshowshスクリプトを実行できるようになる。この状態のスクリプトは通常のコマンドと変わるところがなく、/usr/local/bin、/binなどに格納しておけば、コマンド名（showsh）だけで実行することができる。

```
bash$ ./showsh
/bin/bash
```

今行った作業を解説しよう。まず「#!」で始まる1行だが、これはこのスクリプトファイルを実行する処理系（シェル）を指定している。ドットコマンドやsourceコマンドを使った場合は、bashでスクリプトが実行されることが保証されているが、こちらの方式で指定するのはコマンド名だけなので、bashで実行することを明示的に指定する必要があるからだ。ちなみに、cshのスクリプトを書くときには、これが

```
#!/bin/csh
```

perlのスクリプトを書くときには

```
#!/usr/bin/perl
```

になる。

次の“chmod +x”はファイルに実行属性を付けるコマンドだ。C言語などで開発したコマンドはコンパイル時にコンパイラが実行属性を自動的に設定するが、スクリプトはただのテキストファイルにすぎないので、ユーザーが明示的に実行属性を指定しなければならない。

この設定を施した後も、ドットコマンドやsourceコマンドでshowshスクリプトを実行することは可能だ。シェルスクリプトではシャープ（#）から行末まではコメントとして扱われ、実行時には無視されるので「#!」行はなんの影響も及ぼさない。

このようにしてユーザーが作成したシ

エルスクリプトをコマンドとして利用できるようになる。が、実際にやってみるとshowshを実行しても“command not found”が表示されるだけで、“./showsh”と明示的にカレントディレクトリのshowshであることを指定しないと実行できないことに気付くだろう。これは、コマンド検索パスを設定するシェル変数PATHには、初期状態ではカレントディレクトリが含まれていないためだ(シェル変数については後述する)。LinuxではDOSやWindowsなどとは異なり、明示的に指定しない限り、カレントディレクトリがコマンド検索パスとして扱われることはない。カレントディレクトリにあるコマンドを名前だけで実行できるようにしたければ、コマンドラインから“PATH=\$PATH:.”を実行すればよいのだが、マルチユーザーOSであるLinuxでは、これは危険であるとされている。もし/tmpなど誰でもファイルが作成できる場所に、lsのように頻繁に使われるコマンドと同じ名前で、大事なファイルを消してしまうようなシェルスクリプトを仕込まれ、なおかつ何らかの手段で/bin/lsがリネームされていたら、気付かないうちに/tmp/lsを実行させられてしまう。これは1つの例にすぎないが、カレントディレクトリをコマンド検索パスに含めることは、システムセキュリティ上の問題となる可能性が高い。もしコマンド検索パスにカレントディレクトリを含めるならば、こうした危険性を気に留めておいて欲しい。

以上で基本的なスクリプトの作成方法と実行方法の解説を終えるが、最後に少しだけshowshを改良してみよう。ここまでに作ったshowshコマンドは、実行したユーザーのログインシェルを表示するだけだったが、showshスクリプトに引数としてユーザー名を渡せるようにして、誰のログインシェルでも表示できるようにしよう。このためには、showshファイルを次のように修正する。変更箇所は“\$USER”を“\$1”に修正した部分だけだ。

```
#!/bin/bash
grep ^"$1" /etc/passwd | cut -d:
-f7
```

こうすると、次のようにしてshowshスクリプトを実行して、任意のユーザーのシェルを表示させることができる。このように、シェルスクリプトには、引数を渡すことが可能だ。

```
bash$ showsh keisuk-t
/bin/bash
```

しかしながら、このように単純なスクリプトでは、引数を指定しなかったときには予期しない動作をしてしまうし、存在しないユーザー名を入力したときにはなにも表示されない。コマンドとしての体裁を整えるには、もう少し改良が必要だ。そのためには引数が指定されているかどうか、指定したユーザーが存在するかなどの条件判断をしなければならないので、もう少しシェルスクリプトの文法に関する知識が要求される。そこで、次はbashのスタートアップスクリプトを題材にして、シェルスクリプトの文法を解説する。



スタートアップスクリプトの使い方

冒頭で述べたように、bashを起動するとホームディレクトリに格納された.bash_profileと.bashrcが実行される。ただし、bashの起動方法によって実行されるスクリプトが異なるので、ここで正確な手順をまとめておこう。

まずコンソールのloginプロンプトからユーザー名とパスワードを入力した後に起動された場合は、bashはログインシェルとして起動され、次の順序でスクリプトが実行される。

- /etc/profileがあれば、これが実行される
- 次の優先順位で、最初に見つかったものだけが実行される
 1. ./bash_profile
 2. ./bash_login

3. ./profile

基本的には、2つのシェルスクリプトが実行されるわけだ。一般ユーザーは自分のホームディレクトリに.bash_profileを用意して、ログイン時の設定を行うと考えておけばよい。/etc/profileは管理者がすべてのユーザーに実行してもらいたいコマンド(環境変数の設定など)を記述するためのものだ。なお、このとき暗黙のうちには.bashrcが実行されないことに注意していただきたい。

ログインシェルではない場合には、単純にホームディレクトリの.bashrcだけが実行される。ログインシェルでない状態でbashが起動されるケースには、コマンドラインからユーザーがbashを起動する、ktermを起動する、suコマンドでユーザーを切り替えるなどがあげられる。基本的には、ユーザー認証直後に起動されるシェルはログインシェルであり、それ以外はログインシェルではないと考えておけばよい(suコマンドは例外だが)。ただし、“bash -login”、“kterm -ls”、“su --login”のように、いずれもオプションによってログインシェルとして起動することができる。

ところで少し脱線するが、bashはどのようにしてログインシェルとして起動されたかどうかを判断しているのだろうか。判断基準は2つある。1つは明示的にオプションを指定して“bash -login”として起動された場合、これは分かりやすい。もう1つは“-bash”のように、ハイフンを付けて起動された場合だ。psコマンドを実行すると、COMMAND欄に“-bash”と表示されたプロセスを見つめることができる。これがログインシェルとして起動されたbashである。unixではプログラムを起動するとき、起動するプログラムのファイル名と、起動するプログラム名を個別に指定できるので、これを利用して“/bin/bash”を“-bash”として起動すると、ログインシェルとして起動されるのだ。通常、ユーザーがシェルからコマンドを実行するときには、プログラム

のファイル名とプログラム名は一致して実行されるが、login時にはこうした特殊な方法でシェルが起動されるわけだ。あまり意味はないが、“ln /bin/bash ./bash”としてから“./-bash”を実行すれば、何のオプションをつけなくても、bashがログインシェルとして実行され、この仕組みが確認できる。

話を元に戻そう。bashの起動モードにはログインシェルとそうでないモードがあることを述べたが、もう一つ、これらインタラクティブモードと区別される「ノンインタラクティブモード」がある。これは先に登場したshowshスクリプトのよう

に、シェルスクリプトを実行するためにbashが起動されたときのモードである。この場合は、環境変数ENVが参照され、これに設定されているファイルがスタートアップスクリプトとして実行される。たとえば、

```
export ENV=~/.bashrc
```

とされていれば、ホームディレクトリにある.bashrcが実行される。環境変数ENVが定義されていなければ、何も実行されない。

この仕組みのため、不用意にシェルスクリプトを作ると、あるユーザーの環境

では動作するが、他のユーザーの環境では動作しないということもありうるので注意が必要だ。たとえば、あるユーザーの環境では、上記のように環境変数ENVが定義され、また.bashrcでは環境変数PATHが定義されていたとしよう。ところが他のユーザーでは異なる環境変数PATHが定義されていたため、シェルスクリプト内部で呼び出しているコマンドが見つからず、実行できないということがありえるのだ。シェルスクリプトを作るときには、シェルの環境設定に左右されずに動作するよう考慮しなければならないということである。

デフォルトの スタートアップスクリプト

自分専用のスタートアップスクリプトを作る前に、まずはサンプルとして用意されているスタートアップスクリプトを参考にしてみよう。ここでとりあげるのは、RedHat 5.2のadduserコマンド（これもシェルスクリプトである）でユーザーを作成したときに、ホームディレクトリに用意されるスタートアップスクリプトだ（リスト1~3）。これらのファイルは/etc/skelに格納されているので、最初からカスタマイズしたスタートアップスクリプトを配りたければ、ここのファイルを編集すればよい。

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin
ENV=$HOME/.bashrc
USERNAME=""

export USERNAME ENV PATH
```

ホームディレクトリに.bashrcが通常ファイルとして存在すれば、.bashrcを実行する

コマンド検索パスを設定するシェル変数。環境変数としても意味を持つ

ノンインタラクティブモードで実行されるスタートアップスクリプトを指定するシェル変数

ユーザー名を参照するために汎用的に使われる環境変数。デフォルトでは空なので、自分のユーザー名を指定する

定義したシェル変数を環境変数として宣言する

リスト1 デフォルトの.bash_profile

演算子	真となる条件
ファイルテスト	
-d <ファイル>	<ファイル> がディレクトリ
-e <ファイル>	<ファイル> が存在している
-f <ファイル>	<ファイル> が通常ファイル
-L <ファイル>	<ファイル> がシンボリックリンク
-r <ファイル>	<ファイル> の読み込みが許されている
-s <ファイル>	<ファイル> のサイズが0より大きい
-w <ファイル>	<ファイル> への書き込みが許されている
-x <ファイル>	<ファイル> を実行できる
<ファイル1> -nt <ファイル2>	<ファイル1> が<ファイル2> よりも新しい
<ファイル1> -ot <ファイル2>	<ファイル1> が<ファイル2> よりも古い
文字列テスト	
-z <文字列>	文字列の長さが0
-n <文字列>	文字列の長さが0より長い
<文字列1> = <文字列2>	文字列1と文字列2が等しい
<文字列1> != <文字列2>	文字列1と文字列2が異なる
論理演算	
! <条件式>	条件式が偽のとき
<条件式1> -a <条件式2>	条件式1と条件式2が両方真のとき
<条件式1> -o <条件式2>	条件式1と条件式2のどちらかが真のとき

表1 条件式で利用可能な演算子

if文

まずはログインシェルとしてbashが起動されたときに実行される、.bash_profileを先頭から見ていこう（リスト1）。1行目はシャープ（#）で始まっているので、すでに述べたように単なるコメント文である。showshスクリプトでは先頭行に「#!/bin/bash」を記述したが、.bash_profileと.bashrcはbash専用のシェルスクリプトなので必要ない。

もう一つのコメント行を飛ばして、次が「if」から「fi」までのブロックで構成されるifブロックである。ここでの処理内容は、「もしホームディレクトリに.bashrcがあれば、これを実行する」というもの

だ。

if文の書式は次のようになっている。

```
if [ <条件式> ]; then
    <条件式が成り立ったときに実行
      するコマンド>
fi
```

リスト1の場合は“-f /.bashrc”が条件式であり、fは「指定したファイルが存在し、かつ通常ファイルならば真、存在しなければ偽」となる演算子である。指定したファイルが存在してもディレクトリならば、偽となる。fと同じようにif文の条件式に利用可能な演算子のうち、利用頻度の高いものを表1に示す。

ifブロック内部のコマンドである“. /.bashrc”はすでに解説したとおり、ホームディレクトリにある.bashrcを実行するという意味だ。つまり、bashがログインシェルとして実行されたときには.bash_profile、そうでないときは.bashrcというようにスタートアップスクリプトは分かれているが、どちらでも実行したい内容は.bashrcに記述して、ログイン時のみ実行したい内容を.bash_profileに記述する、という方針を実現しているわけだ。通常ログインシェルであろうがなかろうが、bashが起動したときに行う環境設定に大きな違いはないので、.bash_profileにこの3行を加えておくのはセオリーとなっている。一般的に.bash_profileだけで実行する内容は、次に解説する環境変数の設定と、ターミナルの設定ぐらいのものだ。

以上で.bash_profileのifブロックの解説は終わりだが、もう少しif文の解説を続けよう。

条件式を囲うブラケット([])の後ろ

にあるセミコロン(;)がいささか奇妙に思えないだろうか。しかし、これには訳があるのだ。このセミコロンは条件式の終端を表すためにここにあるのだ。もし省きたければ、次のように記述することも可能だ。

```
if [ <条件式> ]
then
    <コマンド>
fi
```

しかし、条件式の終わりならば、ブラケットで十分とも言える。これを説明するためには、より正確にif文の動作を解説する必要がある。実はif文の正確な構文は次のようになっている。

```
if <コマンド> ; then
    <コマンドが正常終了したときに
      実行するコマンド>
fi
```

つまり、ifとthenの間には、lsやgrepなど、いかなるコマンドでも指定できるのである。Linuxのコマンドは正常終了したとき0、それ以外のときに1~255の値で表される終了ステータスを発行する(コマンドの作りにまかされている慣習にすぎないので、すべてのコマンドがそうであるとは限らないが)、この終了ステータスが0の場合に真として扱うのが、if文である。

終了ステータスは“echo \$?”を実行すると見ることができるので、lsコマンドで試してみよう。

```
bash$ ls /tmp          ->正常終了
bash$ echo $?
0
bash$ ls /tmp/hoge     ->異常終了
```

```
ls: /tmp/hoge: No such file or
directory
bash$ echo $?
1
```

if文の条件式にコマンドを指定できることがわかれば、条件式の終端を表すセミコロンは、実はコマンドの区切りに使われるセミコロンと同じであることがわかる。たとえば、次のように2つのコマンドをセミコロンでつなげば、1行で2つのコマンドを順に実行するさせる指示が可能だ。if文もこれと同じというわけだ。

```
bash$ make depend; make zliilo
```

ということはブラケットもプログラムかという、これはビルトインコマンド(built-in command)と呼ばれるbash内蔵コマンドなので、“[”という実行ファイルがなくとも動作する。ただし、bashの元になったシェルであるshでは実際に/usr/bin/[が使われていたので、Linuxのディストリビューションにもまだ存在はしている。余談になるが、/usr/bin/[と同等のコマンドに/usr/bin/testというものもある。ちょっとした自作のプログラムにtestという名前をつけたが、いくらtestコマンドを実行してもうまくいかない、というのは誰もが一度は経験する失敗の1つである。

環境変数とシェル変数

ifブロックの次では、PATH、ENV、USERNAMEという3つのシェル変数を定義している。シェル変数とはシェルが管理する変数のことで、ユーザーが指定した変数名で、文字列を保存しておくことができる。シェル変数の用途は主に3つあ

変数名	引数	機能
PATH	コロンで区切ったパスリスト	PATHに登録されたパスにあるコマンドは、コマンド名だけで起動できる
ENV	ファイル名	ノンインタラクティブモードでbashが起動されたときに、スタートアップスクリプトとして読み込まれる
HOME	パス	ホームディレクトリを指定する。引数なしのCDコマンドで移動するディレクトリ
CDPATH	コロンで区切ったパスリスト	CDPATHに登録されたパスにあるディレクトリは、ディレクトリ名だけで移動できる
HISTSIZE	数値	保存しておくヒストリの個数
HISTFILESIZE	数値	ヒストリファイル(デフォルト:.bash_history)の最大サイズ
PS1	表3を参照	プロンプトを指定する
IGNOREEOF	なし	定義されているとEOFキャラクタ(デフォルト:Ctrl-D)ではシェルを終了しなくなる
MAIL	ファイル名	メールスプールを指定しておく、定期的にメールの着信チェックが行われる

表2 bashの組み込みシェル変数

る。1つは複雑なシェルスクリプトを作る際に、値の保存に使う。もう1つはbashが使用する組み込みシェル変数に値を設定してシェルをカスタマイズしたりシェルの状態を取得する。最後の1つは環境変数を定義するためだ。

シェル変数を定義するには、次の構文を使用する。シェル変数名に使用する英字の大文字小文字は区別される。

```
<シェル変数名> = <文字列>
```

シェル変数を参照するには、「\$<シェル変数名>」とする。参照したシェル変数が定義されていなくても、空文字列に置き変わるだけでエラー扱いにはならない。次に簡単な使用例を示す。

```
bash$ USERNAME="keisuk-t"
bash$ echo $USERNAME
keisuk-t
```

以上が簡単なシェル変数の使い方である。シェル変数はこうしてユーザーが自由に定義して利用できるが、なかにはbashの組み込みシェル変数として予約されている名前もある。主なbashの組み込

みシェル変数は表2に示す。これらのシェル変数は定義すると自動的にbashによって参照され、特殊な用途に利用される。bash_profileで定義しているシェル変数では、PATHとENVが組み込みシェル変数である。また逆にbashによって自動的に定義されるシェル変数もある。これらには、参照するたびに乱数値を返すRANDOMやカレントディレクトリを返すPWDなどがある。また先に登場した?(\$?で参照し、直前に実行したコマンドの終了ステータスを返す)もこの類のシェル変数の1つである。

bash_profileでは、シェル変数を定義した後、それぞれの変数についてexportコマンドを実行している。シェル変数にexportコマンドを適用すると、そのシェル変数は環境変数としても機能するようになる。シェル変数はbashによって管理されているためbashが利用するだけだが、環境変数はOSによって管理され、あらゆるアプリケーションから参照できるようになる。したがって、どのような環境変数を定義するかは、利用するアプリケーション次第となる。設定しておくとも便利

な環境変数には、エディタを設定しておくEDITOR(例:/usr/local/bin/emacs)やlessコマンドによる漢字コード変換ルールを設定するLESSCHARSET(例:japanese-ujis)などがある。なお、exportコマンドを適用しても、その変数はシェル変数と環境変数の両用変数となるので、その後もシェル変数として参照可能である。

シェル変数と環境変数には、もう1つ大きな違いがある。それは、シェル変数は設定したbashプロセス以外からは参照できないが、環境変数は設定したbashプロセスから起動された他のプロセスからも参照可能ということだ。つまり、bash(bash Aとする)からさらにbash(bash Bとする)を起動したとき、bash Aで定義したシェル変数はbash Bでは参照できないが、bash Aで定義した環境変数はbash Bでも参照できる。もちろん、bashからemacsやperlなどのアプリケーションを実行した場合に参照できるのは環境変数だけだ。

この変数の違いはシェルスクリプトを作るときに、少なからず影響を及ぼす。次の内容のシェルスクリプト「showvar」を作って実験してみよう。showvarスクリプトにはchmodコマンドを使って実行属性をつけておくこと。

```
#!/bin/bash
echo $SHELLVAR
echo $ENVVAR
```

次のように、showvarスクリプトをドットを使った方法とコマンドライクな方法で実行すると、結果が異なることが確認できる。

```
bash$ SHELLVAR=shellvar-defined
->シェル変数
bash$ export ENVVAR=envvar-defined
->環境変数
bash$ . showvar
shellvar-defined
envvar-defined
bash$ showvar

envvar-defined
```

```
# .bashrc

# User specific aliases and functions

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

/etc/bashrcが通常ファイルとして存在すれば、/etc/bashrcを実行する

リスト2 デフォルトの.bashrc

```
# /etc/bashrc

# System wide functions and aliases
# Environment stuff goes in /etc/profile

# For some unknown reason bash refuses to inherit
# PS1 in some circumstances that I can't figure out.
# Putting PS1 here ensures that it gets loaded every time.
PS1="[u@h \W]\\$ "

alias which="type -path"
```

bashのコマンドプロンプトを設定するシェル変数。書式は表3を参照

プログラムが格納されているディレクトリを調べる" type -path " コマンドをエイリアスwhichに定義する

リスト3 /etc/bashrc

bash\$

ドットで起動したときには両方の変数が表示されるが、コマンドとして実行したときには、環境変数だけが表示されていることがわかる。これは、両者の方法でスクリプトの起動方法が異なるための結果だ。ドットを使った場合、現在のbashがスクリプトを読み込み、実行する。このためシェル変数と環境変数の両方を参照できる。一方コマンドとして実行したときには、「#!」で指定したプログラム、つまりbashが新たに起動され、このbashでスクリプトが実行される。このため環境変数だけが参照可能であり、シェル変数は未定義と解釈され、空文字列が表示された。

シェルスクリプトの作成中はドットで起動してテストし、完成したらコマンドにしよう、などと考えていると思わぬトラブルに見舞われる可能性もあるので注意したい。

● .bashrc

次に.bashrcを見てみよう。このスタートアップスクリプトは本来ログインシェルではないときに実行されるファイルだが、.bash_profileから明示的に読み込まれているため、ログインシェルの場合でも実行されるのはすでに解説したとおりである。

その内容はリスト2のとおりだが、単に/etc/bashrcが通常ファイルとして存在すれば、それを実行する、としているだけで特に解説すべき事柄はない。

そこで/etc/bashrcを見てみることにしよう(リスト3)。このファイルはログインシェルでないときに、すべてのユーザーに実行してもらうためのスクリプトだが、/etc/bashrcを編集して利用する場合には、.bash_profileや.bashrcと違って、bashが暗黙のうちに実行するわけではないことを考慮しておく必要がある。

シェル変数PS1

/etc/bashrcでは、シェル変数PS1の定

義とエイリアスwhichの定義を行っている。シェル変数PS1は表2にも示したように、bashの組み込みシェル変数である。このシェル変数に文字列を定義しておく、それがシェルのプロンプトとして表示される。たとえば、

```
PS1="bash$ "
```

と定義すると、コマンド入力待ちの状態になると“bash\$”が行頭に表示される。また表3に示す特殊な文字列をシェル変数PS1に含めると、いつも決まりきった文字列ばかりではなく、カレントディレクトリやログイン中のホスト名といった、現在のユーザー環境をコマンドプロンプトの一部として表示できる。/etc/bashrcのように

```
PS1="[\u@\h \W]\\$ "
```

とすれば、ユーザーkeisuk-tがLinux.ascii.co.jpのホームディレクトリで作業中の場合、

```
[keisuk-t@Linux keisuk-t]
```

というプロンプトが表示されることになる。/etc/bashrcのプロンプト設定は筆者としてもおすすめだ。プロンプトに時計を含めたり、複数行に渡る情報量豊かな邪魔くさいプロンプトにすることもできるが、それはお好みでということだ...

エイリアス

シェルスクリプトにするほどではないが、いちいちタイプするのは面倒なときに、オプションを含めてコマンドに別名をつけることができる。これがエイリアスと呼ばれる機能で、次の書式で定義する。

```
alias <エイリアス名> = <置換後の文字列>
```

/etc/bashrcにあるように

```
alias which="type -path"
```

と定義されていれば、“which ls”と入力するだけで“type -path ls”と入力した場合と同じ結果が得られる。“alias ls="ls --color”や“alias rm="rm -i”

などがポピュラーなエイリアスの定義例である。

エイリアスによる置換はコマンドラインの先頭だけに適用されるので、エイリアスlsが定義されていても、“/bin/ls”と入力すれば、これにはエイリアスは適用されない。また、置換後の文字列に置換前のエイリアス名が含まれていても適用されないので、無限に置き換えが行われることもない。

/etc/bashrcで定義されているエイリアスwhichは、引数に指定したコマンドを環境変数PATHから検索し、ありかを表示するコマンドとして利用できる。

```
bash$ which ls
/bin/ls
```

実はまったく同じ機能を持つコマンドが/usr/bin/whichに存在しているのだが、bashの組み込みコマンドであるtypeコマンドを使った方が若干ながら実行速度で有利なため、エイリアスとして定義されている。しかし、/usr/bin/whichとtype -pathはまったく同じ動作をするわけではない。このエイリアスを使って“which which”としても、何も表示されないのだ。これは“type -path <コマンド名>”が、<コマンド名>をプログラムファイルとして見つけた場合には、パスを含めて表示するが、エイリアスとして見つけた場合には何も表示しないからだ。昔ながらのwhichコマンドの動作

コマンド	機能
%%t	現在時刻(時:分:秒)
%%d	現在日時(曜日 月 日)
%%n	改行
%%s	シェル名前(bash)
%%w	現在の作業ディレクトリ(フルパス)
%%W	現在の作業ディレクトリ(ベース名のみ)
%%u	ユーザー名
%%h	ホスト名
%%#	コマンド番号
%%!	ヒストリ番号
%%\$	ルートユーザーの場合「#」、それ以外の場合は「\$」
%%nnn	ASCIIコードを8進数で表した文字
%%#	バックスラッシュ

表3 プロンプトに使用可能な特殊文字列

を期待するならば、このエイリアスはあまりいい考えではない。素直に/usr/bin/whichを使った方がいいだろう。なお、“type which”とすれば、whichがプログラムファイルかエイリアスかを調べることができる。

● /etc/profile

スクリプトがスクリプトを呼び出す順序に従って、.bash_profile、.bashrc、/etc/bashrcと追いかけてきたが、これでひとまず終了である。そこで最後に、解説を後回しにしておいた、/etc/profile(リスト4)を解説しよう。内容が他のスクリプトに比べて複雑だったので解説を後回しにしたが、実際にはログインシェル起動時に.bash_profileよりも先に実行されるスクリプトファイルである。

コマンド置換

/etc/profileの細かな部分はずでに解説されているので、ここでのトピックは2つに絞る。1つはif文の条件式に使われている、バッククォート(`)によるコマンド置換機能である。

このif文での処理は「ログインしたユーザーのユーザー名とユーザーグループ名が同じで、かつユーザーIDが14よりも大きければumask 002を、そうでなければumask 022を実行する」というものだ(図1: umaskの設定方法を参照)。この処理からはRedHatのユーザー管理ポリシーを読み取れる。そのポリシーとは次のようなものだ。ユーザーを新規作成するときには、ユーザーと同じ名前のグループも同時に作成し、基本的にユーザーとグループは1対1に対応させておく。こうすれば他のユーザーのファイルにアクセスしたいときには、目的のユーザーに対応

しているグループに所属すればいい。ただし、利用頻度が低いユーザーなどには専用のグループは作らず、まとめて1つのグループで管理する。この場合は同じグループに属していても、互いに書き込みできないumaskが設定される。こんなところだ。

ただし、ユーザーIDが14以下のユーザーはrootやshutdown、ftpといった管理用ユーザーアカウントとしてあらかじめ使用されているので、この場合は特例として該当ユーザーのみが書き込みを行えるumaskを設定する。

もっとも、これは1つの指針にはなるが、必ずしもこのポリシーに従う必要はない。自分が作成したファイルが他人に見られては困るというのなら、.bash_profileに“umask 077”を書き加えればよい。この辺は、サイトごとの管理ポリシーにしたがってumaskを設定すればよいだろう。

このif文の条件式で使われている機能がコマンド置換機能である。

`<コマンド>`

のようにコマンドをバッククォートで囲むと、そのコマンドを実行した結果、標準出力に出力された文字列が、コマンドラインに展開される。たとえば、ユーザーkeisuk-tにはユーザーID「500」が割り当てられ、ユーザーグループ「users」に属しているとしよう。すると、コマンドidの実行結果は次のようになる。

```
bash$ id -un      ->ユーザー名を表示する
keisuk-t
bash$ id -gn      ->グループ名を表示する
uesrs
bash$ id -u       ->ユーザーIDを表示する
500
```

この場合、/etc/profileのif文の条件式は次のようになる。

置換前:

```
if [ `id -un` = `id -gn` -a `id -u` -gt 14 ]; then
```

置換後:

```
if [ keisuk-t = users -a 500 -gt 14 ]; then
```

```
# /etc/profile

# System wide environment and startup programs
# Functions and aliases go in /etc/bashrc

PATH="$PATH:/usr/X11R6/bin"
PS1="[u@h W]\\$ "

ulimit -c 1000000
if [ `id -gn` = `id -un` -a `id -u` -gt 14 ]; then
    umask 002
else
    umask 022
fi

USER=`id -un`
LOGNAME=$USER
MAIL="/var/spool/mail/$USER"

HOSTNAME=`/bin/hostname`
HISTSIZE=1000
HISTFILESIZE=1000
export PATH PS1 HOSTNAME HISTSIZE HISTFILESIZE USER LOGNAME MAIL

for i in /etc/profile.d/*.sh ; do
    if [ -x $i ]; then
        . $i
    fi
done
```

デフォルトで設定されている /usr/local/bin:/bin:/usr/binに /usr/X11R6/binを追加する

プログラムが異常終了したときに作成されるcoreファイルの上限サイズを指定する。無駄にディスクスペースが使われるのを抑制する

ユーザー名とグループ名が等しく、かつユーザーIDが14よりも大きい場合、umask 002を、そうでない場合はumask 022を実行する

bashの組み込みシェル変数の定義

/etc/profile.dディレクトリに作成された*.shスクリプトファイルをすべて実行する

リスト4 /etc/profile

for文

もう1つのトピックがfor文である。ここでの処理は、/etc/profile.dディレクトリにある、“*.sh”にマッチするファイルをすべて、順にシェルスクリプトとして実行するというものだ。for文は次の書式で利用する。<パラメータリスト>にはスペースで区切った文字列を0個以上並べることができる。なお、for文の後の<シェル変数名>にはダラー(\$)をつけず、<パラメータリスト>の終わりのセミコロンを忘れないようにすること。

```
for <シェル変数名> in <パラメータリスト>; do
    <繰り返し行う処理>
done
```

for文によって、<パラメータリスト>に指定した文字列が先頭から順に<シェル変数名>に代入され、<繰り返し行う処理>が実行される。/etc/profileのfor文ならば、/etc/profile.dにinit.shとenvironment.shの2つのファイルがあった場合、まずシェル変数iにenvironment.sh(アスタリスク(*)の展開はASCIIコード順に行われる)が代入され、このファイルに実行属性(chmod +x)が付けられているのを確認してからドットで実行される。次にinit.shが代入され、同じようにもう一度処理される。

つまりこのfor文は、/etc/profileにはこれ以上書き加えず、追加したい処理があれば/etc/profile.dにシェルスクリプトを作りなさい、というメッセージである。初期状態ではこのディレクトリは空なので、何も行われない。なお、/etc/profile.dにシェルスクリプトを追加するときには、chmodコマンドで実行属性を付けるのを忘れないこと。

なお、表4、表5にシェルにおけるワイルドカードの扱い、特殊文字の意味をまとめておく。

以上でbashの起動時に実行されるスタートアップスクリプトはすべて見てきたことになる。現状を把握できたので、後は自分好みに環境をカスタマイズできるはずだ。

今回は引き続きbashのシェルスクリプトをとりあげ、スタートアップスクリプトのカスタマイズを行い、ちょっと便利なシェルスクリプトコマンドを作成する予定だ。

ワイルドカード	マッチ(一致)対象
*	すべての文字にマッチする(空の文字を含む)。ただし、先頭の“(ドット)にはマッチしない。
?	任意の1文字にマッチする。先頭の“(ドット)にはマッチしない。
[キャラクタセット]	キャラクタセットに指定した文字にマッチする。a-zのように文字の範囲でも指定できる(この場合英小文字aからzの文字)。
![キャラクタセット]	キャラクタセットに指定した文字以外にマッチする。

表4 ファイル名の展開の際に利用できるワイルドカード

特殊文字	使用目的	例
;	特殊文字複数のコマンド入力	make dep; make clean; make
&	コマンドをバックグラウンド実行	tar xf file.tar &
()	コマンドの標準出力をグループ化	(command1; command2) command3
	コマンド間をパイプでつなぐ	zcat file1.tar.gz tar xvf -
<	入力リダイレクション	patch -p0 < file1.patch
>	出力リダイレクション	ls -lR > FileList.txt
\$(var)	シェル変数の置換	\${USER}
`コマンド`	コマンド実行結果の置換	UID=`id -u`
¥	特殊文字の働きの打ち消し、コマンド行の最後で使用すると次行が継続行となる	cd a¥¥
'文字列'	文字列の中の特殊文字列の働きの打ち消す	'1,000'
"文字列"	文字列の中の特殊文字列の働きの打ち消す	"\$1=¥140"
#	コメント	#これはコメントです
{コマンド1; コマンド2}	現在のシェルでコマンドを実行	{コマンド1; コマンド2 > コマンド3}

表5 シェルで利用される特殊文字

Linuxのファイルシステムでは、ファイルのオーナー、オーナーの属するグループ、その他のユーザーのそれぞれについて、ファイルの読み込み(r)、書き込み(w)、実行属性(x)を指定できる。なお、ディレクトリの場合は、ファイル一覧表示(r)、ディレクトリ内のファイル作成、削除、リネーム(w)、ディレクトリへの移動(x)を意味する。umaskコマンドは、ファイルやディレクトリの作成時にデフォルトでどの属性を「オフ」にするかを指定する。

オーナー	グループ	その他
r w x	r w x	r w x
1	1	1
2	2	2
4	4	4

↓ フラグをオフにする部分のみ足し合わせる

例：ファイルのオーナー以外には書き込みを許可しない設定

オーナー	グループ	その他
- - -	- w -	- w -
0	0	0
0	2	2
0	0	0
0	2	2

図1 umaskのパラメータ