Ubuntu Magazine Japan ウブントゥ マガジン ジャパン

ご利用になる前に必ずお読みください

このＰＤＦファイルの内容についてのご質問・お問い合わせは株式会社アスキー・メディアワークスでは一切お受けできません。ご自身の責任においてご利用ください。

クレジット：
●文/水野 源（Ubuntu Japanese Team）
●まんが/瀬尾 浩史
●デザイン/シオズミタロウ
●初出/株式会社アスキー・メディアワークス「Ubuntu Magazine Japan vol.04」（http://ubuntu.asciimw.jp/）2010 年 5 月 31 日発行

not scary!! コマンドなんて怖くない not scary!!

はじめてのシェルスクリプト入門

Introduction to shell script

コマンドに少し慣れたらシェルスクリプトに挑戦！

前号の特集「コマンド&端末入門」に引き続き、今回はCLI（コマンドライン・インタフェース）をさらに便利に使うテクニックとして、シェルスクリプトと正規表現、シェルのジョブコントロールの方法を紹介するぞ。なお今回の内容は前回紹介した知識を前提として進めるので、前号を読んでいないバッドボーイズ&メガネガールズの諸君はいますぐ本屋さんにダッシュするか、Ubuntu Magazine Japanのブログ（http://ubuntu.asciimw.jp/）から前号のPDFファイルをダウンロードしておさらいしよう！

そもそも、シェルとかスクリプトってなに？

シェルスクリプトとは、文字通り「シェル上で動くスクリプト言語」のことだ。スクリプト言語というのは、ウェブサイトのCGIプログラム作成でよく使われているPerlのような、プログラミング言語の一種だと思ってほしい。シェルスクリプトに限って言えば、WindowsやMS-DOSで利用されているバッチファイルと同じようなもの、と考えてくれてもOK。テキストファイルに実行したいコマンド（プログラム）をあらかじめ記述しておいて、そのファイル（＝スクリプト）をシェルに読み込ませることで、一連の処理を実行させてしまおうというわけだ。

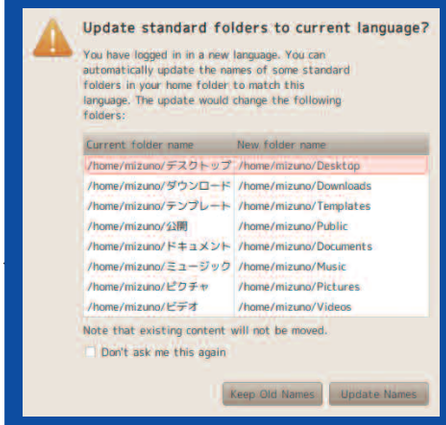
しかもシェルスクリプトは、変数に値を格納して再利用したり、条件によって処理を分岐させたりといったこともできる。つまりとても複雑で柔軟な処理であっても、ラクラクこなすことができるのだ。そしてUnix系のOSではシステムを動かすため、いたるところで様々なシェルスクリプトが活用されている。例えば「/etc/init.d」以下に配置されている伝統的な初期化処理や、cron（決まった処理を定期的に自動実行する仕組み）で実行される処理などはシェルスクリプトで書かれている。他にも日本語環境セットアップヘルパのように、アプリケーションに見えて実はシェルスクリプト、というソフトもある。

シェルスクリプトにも種類がある！

前号でzsh（zシェル）というシェルを少しだけ紹介したので勘のいい人はもう気づいているかもしれないが、ひとくちにシェルといっても様々な種類があり、Unixはユーザがシェルを自由に選べるようになっている。

シェルには大きく分けてsh（Bourneシェル）系というシェルと、csh（Cシェル）系という系列があるのだが、残念ながらこれらの間では完全な互換性はない。UbuntuなどのGNU/Linuxの環境では、shをさらに高機能にしたbash（Bourne-Again Shell）が標準シェルとして採用されているので、本記事ではbash用のシェルスクリプトを解説するぞ。

ディレクトリ名を英語表記に!!



端末をひんぱんに使うと最初に気づくのは、日本語が扱いづらいという点。「デスクトップ」のようなディレクトリへの移動など、いちいち日本語を入力するのは面倒だ。 そこで、ホームディレクトリにある日本語のディレクトリを英語名に変更しよう。具体的には端末から下のコマンドを実行するだけ。するとディレクトリ名をアップデートしてよいかの確認ダイアログが表示されるので、確認したら「Update Names」をクリックしよう。

英語にディレクトリ名を変更

➡ディレクトリの中にファイルが存在すると、名前のアップデートは行われない。あらかじめ空にしておくか、インストール後すぐにやってしまおう。

```
$ LANG=C xdg-user-dirs-gtk-update
```

●文/水野源（Ubuntu Japanese Team） ●マンガ/瀬尾浩史

いよいよシェルスクリプトを書いてみよう！

スクリプトの編集と実行のしかた

シェルスクリプトを実行するには、あらかじめテキストファイルでスクリプトを書いておく必要がある。ちなみにシェルスクリプトの内容を1行ずつコマンドラインから手で入力することもできるのだが、非常に効率が悪いのはすぐにわかってもらえると思う。

というわけで、さっそく好きなテキストエディタを起動して、**リスト01**のようにスクリプトを入力して保存してみよう。

ファイル名は好きな名前をつけてかまわないが（ここでは「example1.sh」とした）、拡張子は「.sh」にしておくと解りやすい。別に拡張子をつけなくてもスクリプトの動作に影響はないのだが、このような拡張子をつけておけば人間がファイル名を見たときに、一目でシェルスクリプトだと判別できるので都合がいいのだ。保存が終了したら、**リスト02**のように端末からスクリプトを実行しよう。シェルのプログラム（つまりUbuntuではbash）に引数としてスクリプトのファイル名を指定してあげればスクリプトが動き出すぞ。

それでは**リスト01**のスクリプトは何をさせようとしていたのかを説明しよう。これは、列挙されたコマンドを上から順番に実行していくだけの、もっとも単純なスクリプトの例だ。

まず1行目は「date」コマンドで、現在の時刻を表示させている。2行目と4行目では「echo」コマンドを引数なしで実行して、「date」、「who」、「uname -r」コマンドそれぞれの出力の間に空の行を挿入して、見やすく整形している。3行目は「who」コマンドで、現在ログインしているユーザの情報を表示している。5行目ではunameコマンドで、現在動作しているカーネルのバージョンを表示している。

このようにコマンドをシェルスクリプトに列挙しておけば、本来なら複数行の入力が必要なコマンドを、ひとつのスクリプトで実現することが出来るのだ。

List01 コマンドを列挙

```
date
echo
who
echo
uname -r
```

▲コマンドをずらずらと並べるだけでも、シェルスクリプトと言えるのだ。

List02 コマンドを列挙

```
$ bash example1.sh
2010年　4月　9日 金曜日
16:45:01 JST
▲dateコマンドの実行結果

▲echoコマンドで空行表示
mizuno    tty7
2010-04-09 15:51 (:0)
mizuno    pts/0
2010-04-09 15:52 (:0.0)
mizuno    pts/1
2010-04-09 15:52 (dhcp-
050.localdomain)
▲whoコマンドの実行結果

▲echoコマンドで空行表示
2.6.32-16-generic
▲uname -rコマンドの実行結果
```

▲「$ bash example1.sh」でリスト1のシェルスクリプトを実行する。

スクリプト名だけでスクリプトを起動させる

掲示板やアクセスカウンタのようなCGIプログラムのソースを見たことのある人は、プログラムの1行目に「#!/usr/local/bin/perl」のような、「#!」からはじまる行があるのを見たことがあるかもしれない。これはshebang（シェバン）と呼ばれる特別な行。Unixのスクリプトでは、1行目に「#!」からはじまるシェバンを書いておくことで「このスクリプトを動作させるためのプログラム（インタプリタ）を指定する」ことができるようになっているのだ。

では、先ほど作成した「example1.sh」を、**リスト03**のように書き換えてみよう。書き換えが完了したら、**リスト04**のように「example1.sh」に実行権限を与え、実行させてみよう。**リスト02**の実行例ではbashの引数としてスクリプトを渡すことで実行したけれど、今度はスクリプトを独立したコマンドのように実行できているのがわかると思う。シェルによって1行目に書かれたシェバンが解釈され「/bin/bash」が起動してスクリプトを実行してくれているのだ！

シェルスクリプトは、このように実行権限を与えてスクリプト名だけで起動するのが一般的。1行目にはシェバンを忘れずに書くようにしよう。

なお、シェバンで指定できるのはコマンドインタプリタだけではない。例えばシェバンを「#!/bin/cat」としてシェルスクリプトを実行してみよう。シェバンは「自分自身を引数として指定されたプログラムを起動する」という動作をするだけなので、「cat」で自分自身を表示することができる。ちょっとしたトリビアだ。

List03 シェバンを1行目に

```
#!/bin/bash
▲この行を1行目に追加
date
echo
who
echo
uname -r
```

▲シェバンでスクリプト実行のためのインタプリタを指定できるのだ。

List04 実行権限を与える

```
$ chmod +x example1.sh
$ ./example1.sh
```

▲実行権限を与えておけば、スクリプト名だけで起動させられるようになる。

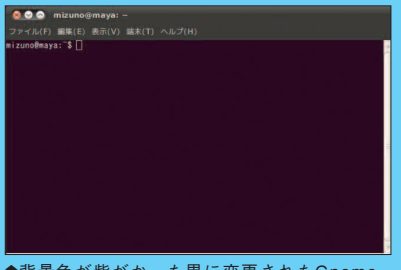
端末のデザインも変わったぞ!!

Ubuntu10.04では標準のデスクトップテーマにLightテーマが採用された。黒と紫をベースカラーとしたLightテーマでは「GNOME端末」の背景色もやや黒っぽい紫色に変更されたのだ。これはデフォルト設定の黒を基調としたAmbianceテーマだけでなく、白を基調としたRadianceテーマでも端末の背景は同様に紫になっているぞ。もし違う色を使いたければメニューの［編集］-［プロファイルの設定］、［色］を開き、好きな色に変更してみよう。プロファイルの設定ではフォントも変えられるぞ。

色やフォントを変更

▲色やフォントの変更はプロファイルを編集して好みの設定を見つけよう！

背景色が変わったのだ



▲背景色が紫がかった黒に変更されたGnome-Terminal。

コマンドが存在するか、検索されるパス

リスト04では、実行権限をつけた「example1.sh」を実行する際、カレントディレクトリを表す「./（ドットスラッシュ）」をスクリプト名の前に付けて、相対パスでスクリプトを指定しているのに気づいただろうか？ 例えば「date」コマンドのプログラムは「/bin/date」にあるのに、コマンド名だけで驚くべきことに相対パスも絶対パスも指定せずに)実行できる。この違いはなんだろう？ その秘密は「コマンド検索パス」という概念にあるので、簡単に説明しよう。

まず端末でリスト05のように入力して、環境変数「PATH」を調べてみよう。環境変数とは、ある環境においてプロセスが参照する設定がされている変数だと思って欲しい。Unixにおいて「PATH」という名前の環境変数には、シェルがプログラムを探す、複数のディレクトリのパスが、コロンで区切られて設定されている。端末から「date」コマンドが入力された時、シェルはこれらのディレクトリを順に検索して「date」というコマンドを探すのだ。「date」コマンドは前述の通り「/usr/bin」に存在するため、パス名を省略して呼び出すことが出来るというわけだ。逆に「example1.sh」のあるユーザのホームディレクトリは「PATH」に含まれていないため、呼び出すときは絶対パスか相対パスを使う必要があるというわけ。

だがここにも落とし穴がある。環境変数の値は簡単に変更できるので「PATH」の値が変更されている状態でコマンドを実行した場合、想定しているプログラムが正しく呼び出せる保証はない。言い換えると、悪意のあるプログラムが「ls」や「cp」のような名前でシステム上のどこかに設置されており、そこにパスが通されていたらどうなるだろう？ シェルスクリプトはどのような状態で実行されるか事前にわからないので、環境変数の値を暗黙的に信頼してしまうのは危険だ。そのため、シェルスクリプト内では、コマンドは絶対パスで呼びだすなどの対策を考慮する必要があることを覚えておいて欲しい(つまり前述のスクリプトの例は、あまり安全じゃないということだね)。

List 05 環境変数PATH

```
$ echo $PATH
/usr/local/sbin:/usr/local
/bin:/usr/sbin:/usr/bin:/s
bin:/bin:/usr/games
```

■このディレクトリに含まれるプログラムはパスを省略して呼び出せる。

グルー言語ってなんだ？

「シェルスクリプトっていったって、単にコマンドを順番に呼び出してるだけじゃん」と思ったかもしれない。そう、まさしくその指摘は正解。前号でパイプを紹介した時に説明した通り、Unixでは単純な機能をもったコマンドを組み合わせて複雑な処理を行うのが基本。シェルスクリプトもそれと同じで、外部のコマンドを結びつけ、そこに簡易な制御機能を足すことで処理を行うように作られている。このように機能そのものを実装するわけではなく、機能の組み合わせに特化して作業を行いやすくする言語を「グルー言語」などと呼んだりする。グルーとは接着剤の意味なので、「既にある部品同士をくっつける」言語というわけだ。PerlやRubyやPythonなどは完全なプログラミング言語だけど、グルー言語としての側面も持っているぞ。

シェルスクリプトでプログラミング

単にコマンドを列挙するだけじゃ面白くない。ここからは変数や制御構造を使って、よりプログラミング言語らしく処理を行う方法を紹介しよう。

変数を使って値を代入・参照させよう

プログラムを組む上で大事なのが、値を記憶しておく変数だ。シェルにもシェル変数という機能があり、プログラマは自由に値を代入したり、参照したりすることができる。リスト06は、「name」という変数に「Ubuntu」という文字列を代入し「echo」コマンドの中で変数を参照している例だ。変数への代入は「変数名＝値」という形で記述する。ここで注意するのは、イコールの両側にスペースを入れてはダメという点だ。C言語などのプログラミング言語の経験がある人は注意しよう。また、複数のワードを代入したい場合は引用符でくくる必要がある(リスト07参照)。

変数を参照するときは、変数名の前に$記号をつければよい。これで「$name」にUbuntuが展開されて表示される。ここでも引用符は大事。ためしに「echo」コマンドの引数を囲んでいる引用符を、単一引用符「'」（シングルクォートとも言う）に変えてみよう(リスト08)。このように、単一引用符でクォートされた中では、特殊記号(ここでは$)が評価されず、「$name」という文字列がそのまま表示されてしまうのだ。

次に、変数の後に続けて文字を表示することを考えてみよう。リスト09のように「$name」に続けてLinuxという文字を入れてみた。実行してみるとわかるが、期待どおりの表示が行われない。これはシェルが「$nameLinux」という変数を参照してしまうためで、この変数にはまだ値が代入されていないので何も表示されないというわけだ。もちろん「$nameLinux」という変数が偶然あった場合、まったく違う値が表示されてしまうことになる。これを回避するためには、変数名を＝（波括弧）でくくってリスト10のように記述する必要がある。つまらないミスを防ぐためにも、変数は「${}」でくくるクセをつけておこう。

List 06 変数への代入と参照

```
#!/bin/bash
name=Ubuntu
echo "My name is $name"
<実行結果>
My name is Ubuntu
```

■引用符や空白の有無に気を付けないと、望む結果が得られないのだ。

List 07 複数の語句の場合

```
#!/bin/bash
name='Ubuntu Linux'
echo "My name is $name"
<実行結果>
My name is Ubuntu Linux
```

■複数のワードを変数へ代入するときは単一引用符でくくること。

List 08 引用符の中の変数

```
#!/bin/bash
name='Ubuntu Linux'
echo 'My name is $name'
<実行結果>
My name is $name
```

■単一引用符の中では変数が展開されないので注意しよう。

List 09 変数名の後ろに文字

```
#!/bin/bash
name='Ubuntu'
echo "My name is
$nameLinux"
<実行結果>
My name is
```

■変数名の後ろに続けて文字があると、何も表示されない。

同じ処理を何度も繰り返すには？

for文でループを使って効率よく！

プログラムを組んでいると、同じ処理を何度も繰り返したくなることがある。ほとんどのプログラミング言語には繰り返しを実現する「ループ構造」が存在するけれど、もちろんシェルスクリプトにもfor文があるぞ。C言語などではforループには終了条件とループカウンタを用いて、指定した回数の繰り返しを行うことが多いが、bashのforループでは反復回数を値のリストとして渡す必要がある（**リスト11**）。具体的な例として**リスト12**を見てみよう。これはUbuntu、Kubuntu、Xubuntuという3つの文字列をリストとして与え、「echo」を実行している例だ。「do」と「done」で囲まれた「echo」が3回実行され、実行されるたびに変数「$in」にリストの各値が順番に代入されているのがわかってもらえると思う。

ここで、前号の75ページを読み返してほしい。前回は複数行をセミコロンで区切って1行にまとめてしまったが、コマンドでこんなこともできるという例として、**リスト13**と同等のコマンドを紹介した。前回は呪文にしか見えなかったコマンドも、今なら何をしているか理解できるのではないだろうか。では中身を見てみよう。

まずfor文のリストには「*.jpg」が指定されている。これは前号でも紹介したワイルドカードの使用例だ。ワイルドカード「*」は任意の文字列を表す記号なので、これは任意のファイル名に「.jpg」という拡張子がついているもの、つまりカレントディレクトリに存在するjpgファイルすべてを意味する。結果としてカレントディレクトリに存在するjpgファイルの回数だけループが実行され、ループが実行される度に変数nにファイル名が代入されるわけだ。そしてループの中では、ファイルの画像フォーマットやサイズの変更を行う「convert」コマンドを実行している。つまりここではカレントディレクトリのすべてのjpgファイルに対し、縮小した画像を作成するという処理を行っているというわけだ。

List 10 変数名の区切りは明確に

```
#!/bin/bash
name='Ubuntu'
echo "My name is
${name}Linux"
<実行結果>
My name is UbuntuLinux
```

▲変数は${}でくくるクセをつけておけば、何も表示されないミスは防げる。

ループ構造はfor以外にもあるぞ

ループには、条件が成立している間だけ繰り返す「while」や、条件が成立するまで繰り返す「until」というものもある。「until」は失敗している間ループし続けるので「コマンドが成功するまで繰り返しを続ける」ような処理を書くことができるぞ。興味があったら調べてみよう！

List 11 同じ処理を繰り返す

```
for 変数名 [in リスト]
do
   処理内容
done
```

▲ループを指定する、for文の構造。

List 13 前号の参考スクリプト

```
#!/bin/bash
for n in *.jpg
do
   convert ${n} -resize 100
${n%jpg}png
done
```

▲前号掲載の、ディレクトリ内のjpgファイルに対してpngのサムネイルを作る例。

List 12 for文にリストを与える

```
#!/bin/bash
for n in "Ubuntu" "Kubuntu"
"Xubuntu"
do
   echo ${n}
done
<実行結果>
Ubuntu
Kubuntu
Xubuntu
```

▲for文に3つの文字列をリストとして与えた場合。

アレな時はコレしたい！ 条件分岐をさせてみよう

testを組み合わせてファイルの条件も指定しよう

その時の状態によって処理を分岐させるというのも、プログラミングではよくあること。というかプログラムなんていうものは、ぶっちゃけるとループと分岐の集合体なのだ。シェルスクリプトで処理を分岐させるには if文や case文を使う。if文の構造は**リスト14**に示した通り。これは条件Aが成立する場合に処理Aが実行され、条件Aが成立しなくても条件Bが成立すれば処理Bが実行され、そして条件AもBも成立しなければ処理Cが実行されるというもの。

リスト15には、もっとも単純と思われる「if」の例をあげてみた。これは「touch」コマンドで hoge ファイルのタイムスタンプを変更し(ファイルが存在しなければ新規に作成する)、コマンドが成功した場合にのみ「echo」が実行されるというわけだ。この例でわかるように「elif」や「else」はオプションで、必ずしも書く必要はないので安心してほしい。

素の状態の「if」では、何かコマンドを実行して終了状態を判定することしかできないが、これに「test」コマンドを組み合わせることでファイルの状態による分岐も行える。**リスト16**はカレントディレクトリに「hoge」という通常ファイルがあるかどうかを調べ、その有無によって表示するメッセージを変化させている。「test」コマンドでは他にも、対象がディレクトリであるか？ パーミッションが与えられているか？ 中身が空でないか？ 2つのファイルのどちらが新しいか？ といったことを調べることができるぞ(**リスト17**参照)。詳しくは「man test」を実行してマニュアルを参照してみてほしい。

「test」コマンドは []（角括弧）という構文に置き換えることができる。この角括弧を使った書き方のほうが一般的なので、覚えておくといいだろう(**リスト18**参照)。ただし変数代入のイコールと違い、括弧の前後にスペースを入れる必要があることに注意しよう。なぜならこの「[（角括弧開き）」が、実は構文ではなくて組み込みコマンドだったりするからだ。実際に呼び出されるのはシェルの組み込み版だが、「/usr/bin」の下をよーく見てみると……！

List14 条件分岐の指定

```
if 条件A
then
  処理A
elif 条件B
then
  処理B
else
  処理C
fi
```

▲if文の基本構造。

List15 もっとも単純なifの例

```
#!/bin/bash
if touch hoge
then
  echo "SUCCESS"
fi
```

▲elifやelseは必ずしも必要というわけではない。こういったシンプルな例もある。

List16 testコマンドを使ったif

```
#!/bin/bash
if test -f hoge
then
  echo "file exist!"
else
  echo "file not found!"
fi
```

▲testコマンドを使ってファイルが存在するか調べられるようになった。

List17 ファイルの状態を指定した条件

```
#!/bin/bash
if test A -nt B
then
  cp A B
fi
```

▲ファイルAがBより新しかったらAをBとしてコピーする、という条件指定。

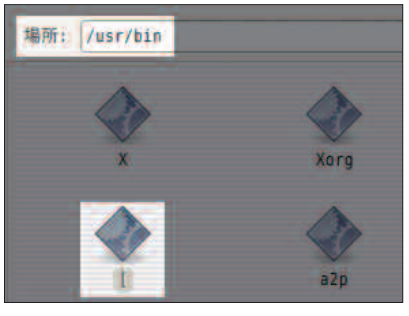
List18 testの一般的な書き方

```
if test -f hoge

if [ -f hoge ]
```

▲上の2つは、まったく同じ条件を指定している。[ ] でくくる方法のほうが一般的だ。

これもコマンド



▲「/usr/bin」の中を見ると、普通に角括弧開き「 [」がコマンドとして存在している。

Caseを使って場合分けをしてみよう

case文はパターンごとの「場合分け」を行う制御文だ。case文の構造は**リスト19**のようになっていて、変数の内容がどのパターンと一致するかを判定し、処理を分岐させることができる。「for」と「case」を組み合わせて、ディレクトリ内のファイルの拡張子ごとに表示するメッセージを変化させている例(**リスト20**)を使って説明しよう。まず全体がforループで囲まれ、リストには「*」が指定されている。これは前ページの**リスト13**と同じ書き方で、カレントディレクトリ内のすべてのファイル分ループすることを表している。そしてループの中のcase文では、変数「file」に対してパターンのマッチングを行っている。パターンのマッチは上から順に行われるため、まず変数「file」が「*.txt」と一致するかが判定される。これはつまり拡張子がtxtかどうか調べているわけだ。もしファイルの拡張子がtxtならパターンにマッチするので「ファイル名 is TEXT FILE」のメッセージを表示して次のループへ移る。「*.jpg」や「*.png」にマッチする場合も同様だ。最後に指定されているパターン「*」は、すべてのファイルにマッチするパターンだ。つまり拡張子がtxt、jpg、pngのいずれでもないファイルは、ここで評価されることになる。

List19 case文の構造

```
case 式 in
  パターン1 ) 処理内容 ;;
  パターン2 ) 処理内容 ;;
  (略)
  パターンN ) 処理内容 ;;
esac
```

▲caseでは、変数の内容によって処理を分岐させられる。

List20 拡張子によって処理を分岐

```
#!/bin/bash
for file in *
do
  case ${file} in
    *.txt ) echo "${file} is TEXT FILE" ;;
    *.jpg ) echo "${file} is JPEG FILE" ;;
    *.png ) echo "${file} is PNG FILE" ;;
        * ) echo "${file} is OTHER FILE" ;;
  esac
done
```

▲forとcaseを組み合わせて、ディレクトリ内のファイルの拡張子を調べている。

コマンドの実行結果で変数やリストを置き換える

便利なコマンド置換で大量のファイル処理も！

変数に値を代入する方法として、イコールを使う方法を紹介した。では変数に、別のコマンドの実行結果を代入するにはどうしたらよいだろうか？ 例として、自分の名前を表示するシェルスクリプトを考えてみよう。自分の名前は「whoami」コマンドで取れるので、素直に考えるとリスト21のようになる。実行してみるとわかるが、変数「myname」には「whoami」コマンドの結果ではなく、「whoami」という文字列が格納されてしまっている。これではダメだ。

このように、コマンド名をコマンドの実行結果で置き換えたい場合は「$()」という構文を使用する。これをコマンド置換と呼び、コマンド置換を使ってリスト21を修正したのがリスト22だ。しかもコマンド置換は変数への代入に限らず、ありとあらゆる場所で使うことができる。その便利っぷりを少し見てみよう。

リスト23は「for」のリストに「find」コマンドの検索結果を与える例だ。カレントディレクトリしか対象にできない「*.jpg」との違いがわかるだろうか？

さらに、リスト24はコマンドの引数に別のコマンドを埋め込んでみた例だ。「mp3gain」コマンドは、引数に複数のMP3ファイルを含むリストを渡すことで、それらのファイルの音量を均一化することができる。そのファイルのリストを作成するのに、「find」コマンドをその場で実行して、結果に置き換えているのだ。こんなにコンビニエンスなコマンド置換、ぜひマスターしてほしい。

List 21 名前の表示に失敗

```
#!/bin/bash
myname=whoami
echo "私の名前は ${myname} です"
<実行結果>
私の名前は whoami です
```

■これは失敗例。コマンドの実行結果ではなく、コマンド名そのものが表示されてしまった。

List 22 表示に成功する書き方

```
#!/bin/bash
myname=$(whoami)
echo "私の名前は ${myname} です"
<実行結果>
私の名前は ubuntu です
```

■$ ( ) でコマンドをくくると、現在のログインユーザ名が表示された。

List 23 jpgファイルへのループ処理

```
#!/bin/bash
for file in $(find . -type
f -iname '*.jpg')
do
  echo ${file}
done
```

■forのリストにfindコマンドの検索結果を与えた例。

List 24 MP3の音量を均一化！

```
$ mp3gain -r -p -k $(find
. -type f -iname '*.mp3' -
printf "%p ")
```

（※mp3gainパッケージのインストールが必要）

■この1行でカレントディレクトリ以下の全MP3ファイルの音量を均一化できる。

算術式展開で計算式を埋め込む

シェル上ではすべての数字が文字として解釈されてしまうため、計算式を記述しても実際に計算式は評価されない。シェルスクリプトで算術演算を行うには、次のような方法を取る。まず一つ目はコマンド置換を用いて「expr」コマンドを使用する方法。引数を計算式として解釈し結果を標準出力へ出力する通常のコマンドで、引数はスペースで区切る必要がある。他にもbashには、文字列を$[]と$(())で囲うことで計算式として評価することもできる。

List 25 シェルスクリプト内での計算方法

```
#!/bin/bash
num=2
echo "$(expr ${num} + 3)"
<実行結果>
5
echo "$[${num}+3]"
<実行結果>
5
echo "$((${num}+3))"
<実行結果>
5
```

■shとの互換性を重視するならexprコマンドを使おう。

bash？ dash？ シェルの拡張構文に注意！

左のリストAのスクリプトでは実行シェルに「/bin/sh」が指定されている。しかしselect文はbashで拡張された構文で、オリジナルのshには存在しないので、スクリプトは動作しないはずだがどっこい、CentOSやFedoraなど、なぜか「動作してしまう」ディストリビューションが存在する。これらのディストリビューションでは「/bin/sh」が「/bin/bash」へのシンボリックリンクになっているため、「/bin/sh」を呼び出しても結果的にbashが呼び出され、bash用のスクリプトが動いてしまうというわけだ。

世の中にはこの例のように、「/bin/sh」を呼び出しているにもかかわらず、「/bin/sh」では動かせない」「でも特定の環境では動いてしまうこともある」スクリプトが少なからず出回っている。これらははっきり言って非常にお行儀が悪い。スクリプトの起動に「/bin/sh」を指定したのであれば、きちんとbourneシェルの機能だけでスクリプトを書くべきだ。

スクリプトを書く際には、自分の使っている構文がどのシェルで提供されているのか、指定するインタプリタは妥当か、という点も考えて作成するようにしよう。

A select文を使った選択メニュー

```
#!/bin/sh
select N in "A" "B" "Exit"
do
  if [ $N = "Exit" ]
  then
    break;
  fi
  echo $N
done
```

■/bin/shが実行シェルだが、存在しない構文selectが使われてしまっている。

お行儀の悪いスクリプト例

bashの拡張構文を使っているため、本来はシェバンを「#!/bin/bash」とすべきスクリプトだ。

B Ubuntu 10.04ではエラー

```
$ ./select.sh
./select.sh: 3: select:
not found
./select.sh: 4: Syntax
error: "do" unexpected
```

■bashで拡張された構文はdashでは実行できない。

C CentOS 5.4では実行できる

```
$ ./select.sh
1) A
2) B
3) Exit
#? 1
A
#? 2
B
#? 3
```

■bashを呼び出すため、本来は実行できないはずのスクリプトが実行できてしまう。

シェルのジョブ制御について知ろう！

サスペンドとレジュームを使う

bashは実行したコマンドを、ジョブという単位で管理していて、複数のジョブを実行したり、自由にジョブの停止や再開、終了などを行うことができるようになっている。ここではジョブ制御機能について紹介しよう。

一度シェルからコマンドを実行すると、コマンドが終了してプロンプトが返ってくるまでユーザは次のコマンドを実行することが出来ない。例えば大きなファイルのダウンロードや、プログラムのコンパイルなど時間のかかる処理を実行していると思って欲しい。そんな時、ちょっと別のコマンドを実行したいんだけど、コマンドが終わるのなんて待てないよ……ということは十分起きうる。CPUに大きな負荷をかけるジョブが実行中の場合は一旦ジョブを停止することで、使用中のリソースを別の作業に回すことができるのだ。**リスト26**のように、現在実行中のジョブを［Ctrl］＋［z］キーを押してサスペンド（一時停止）してみよう。

実行中のジョブがサスペンドされ、プロンプトが返ってくる。こうなれば別のコマンドを実行することができるというわけだ。さて、割り込み作業が終わったらサスペンドしたジョブをレジューム（再開）しよう。レジュームには「fg」コマンドを使用する。これを実行すると、さっきサスペンドしたジョブがまるで何事もなかったかのように再開される（**リスト27**参照）。

なお、複数のジョブをサスペンドしておくこともできる。現在のジョブの状態は「jobs」コマンドで確認しよう。実行中のジョブ番号と各ジョブの状態の一覧に、現在のジョブと直前のジョブには＋と－のマークがついて表示されるぞ（**リスト28**参照）。

フォアグラウンドとバックグラウンド

ジョブには、フォアグラウンドジョブとバックグラウンドジョブがある。フォアグラウンドというのは「いまユーザが操作している／実行しているジョブ」のことだ。例えばnanoエディタを立ち上げれば、フォアグラウンドでnanoが実行されている。それに対し、システムの裏側で非対話的に処理を行うのがバックグラウンドジョブだ。

バックグラウンドでのジョブの実行は、ファイルのダウンロードやコピーのような「時間はかかるが、コマンドを入力した後は放置しておけばよい」タイプのコマンドを実行する際に威力を発揮する。ジョブをバックグラウンドで実行するには、コマンドの最後に「&」をつけて実行しよう（**リスト29**参照）。

フォアグラウンドとバックグラウンドは切り替えることもできる。勘のいい方は気づいたかもしれないが、ジョブのレジュームに使った「fg」コマンドは、フォアグラウンドの意味なのだ。つまり厳密に言えばレジュームしたというより、（サスペンド中のジョブを）フォアグラウンドに持ってきたということになる。そして予想通り、「fg」と対になるコマンドの「bg」が存在する。もちろんこれは、任意のジョブをバックグラウンドで実行させるためのコマンドだ。

リスト30は「cp」コマンドで大きなファイルをコピーしたのだが、思ったよりも時間がかかったため一旦［Ctrl］＋［z］キーでサスペンドし、「jobs」コマンドでジョブのリストを確認した後、「bg」コマンドで「cp」コマンドをバックグラウンド実行に指定した例だ。

ここで一つ疑問が出てきたと思う。「jobs」では3つのジョブが存在するけれど、なぜ「bg」コマンドは的確に「cp」コマンド（ジョブ番号3）に対して働いたのだろう？ それはこの状態では、「cp」コマンドがカレントジョブだったから、というのが回答だ。前述の通り「jobs」コマンドの出力で＋がついているジョブは「カレントジョブ」だ。カレントジョブとは、フォ

List 26 ジョブをサスペンドさせる

```
$ wget
http://releases.ubuntulinux.jp/ubuntu/9.10/ubuntu-
9.10-desktop-i386.iso
<…中略…>
長さ: 723488768 (690M) [application/x-iso9660-image]
`ubuntu-9.10-desktop-i386.iso' に保存中
 0% [                                        ]
6,382,210   4.76M/s                 ^Z
■ここで[Ctrl]+[z]キーを押す
[1]+  停止                  wget
http://releases.ubuntulinux.jp/ubuntu/9.10/ubuntu-
9.10-desktop-i386.iso
```

■処理に時間がかかるコマンドは[Ctrl]+[z]キーで一時停止（サスペンド）できる。

List 27 ジョブをレジューム

```
$ fg
wget http://releases.ubun-
tulinux.jp/ubuntu/9.10/ubun-
tu-9.10-desktop-i386.iso
 1% [
] 11,388,442  4.82K/s    時間
38m 29s
```

■fgコマンドで、続きからダウンロードが再開されている。

List 28 ジョブの一覧を表示させる

```
$ jobs
[1]   停止                  vi
[2]   停止                  nano
[3]   停止                  cat
[4]   停止                  yes
[5]   停止                  vi
[6]   停止                  emacs -nw
[7]   停止                  cat
[8]   停止                  vi
[9]-  停止                  man man
[10]+  停止                  less .bashrc
```

■複数のジョブをサスペンドさせてみた。jobsコマンドでジョブの一覧を表示できる。

List 29 ddを裏で実行

```
$ dd if=/dev/zero of=hoge.img bs=1024k count=1024 &
[1] 2740
$ jobs
[1]+  実行中                dd if=/dev/zero of=hoge.img
bs=1024k count=1024 &
```

■1GBのファイルを作成するddコマンドをバックグラウンドで実行。終了を待たずに次のコマンドを実行できる。jobsで確認すると、裏でジョブが実行中なのがわかる。

List 30 カレントジョブに注目

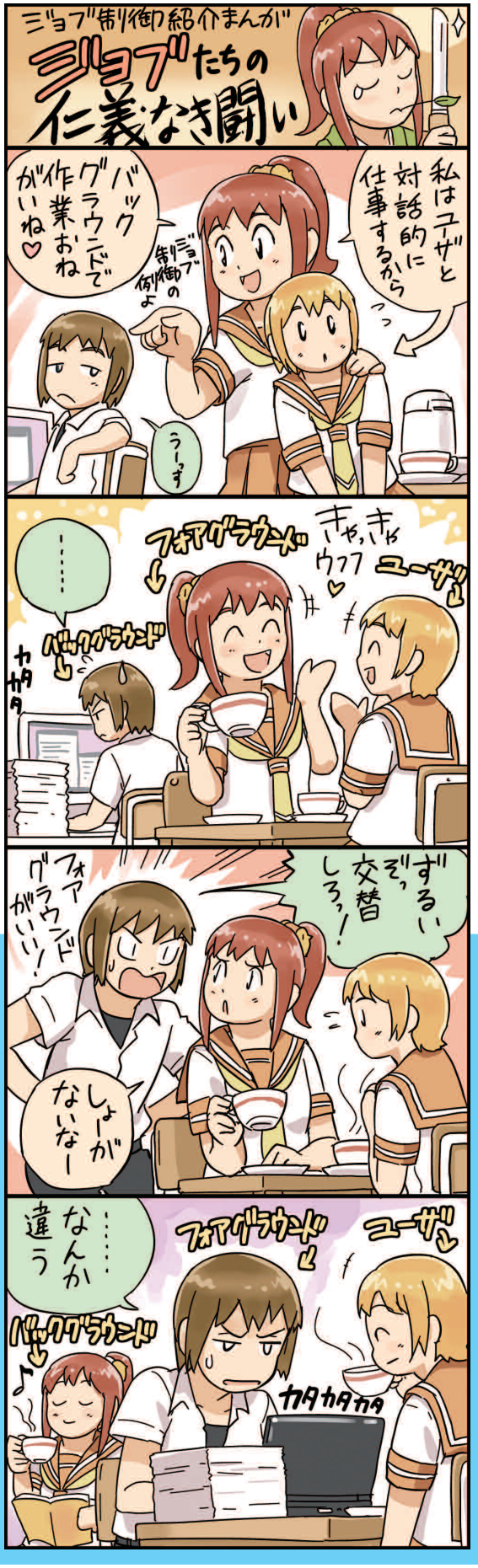
```
$ cp hoge.img fuga.img
^Z
■ここで[Ctrl]+[z]キーを押す
[3]+  停止                  cp hoge.img fuga.img
$ jobs
[1]   停止                  vi
[2]-  停止                  less .bashrc
[3]+  停止                  cp hoge.img fuga.img
$ bg
[3]+ cp hoge.img fuga.img &
```

■ファイルのコピーを一旦サスペンドし、バックグラウンドで再開させている。

List 31 プロセスへシグナル送信

```
$ nano
<nanoエディタ起動>
"fg" とタイプすることで nano を再開できます。
[2]+  停止                  nano
■シグナルを受けてnanoが停止した
```

```
$ ps ux | grep nano | grep -v grep
mizuno    7241  0.1  0.0   3448  1464 pts/0    S+
21:14   0:00 nano
■nanoのPIDを調べる
$ kill -SIGTSTP 7241
■killコマンドでシグナル発行
```



アグラウンドで実行中に停止されたか、バックグラウンドで起動された最後のジョブのことだと思ってほしい。本来「fg」や「bg」はジョブ番号を引数にとるコマンドだが、ジョブ番号を省略した場合は暗黙的にカレントジョブが対象になるのだ(つまりここではbg 3を実行したのと同等だ)。

プロセスを制御するシグナル

ジョブをサスペンドすると、このときシェルの内部では動作中のプロセスにシグナルが送られている。シグナルというのはプロセスが他のプロセスに送るメッセージのことで、[Ctrl]+[z]キーでは通常「SIGTSTP」というシグナルがプロセスに送信されている。「SIGTSTP」というのは、端末からプロセスを中断させるためのシグナルだ。シグナルを受け取ったプロセスは、そのシグナルに応じた動作を行うようプログラムされている(もちろんプログラムの作りによってはシグナルを無視したりもする)。

そしてシグナルを送信するためのコマンドが「kill」だ。名前から「プロセスを殺すコマンド」と誤解されがちだが、実は任意のシグナルを送ることができる(リスト31参照)。

他にもキーボードから[Ctrl]+[c]キーで「SIGINT」シグナルを送信することができる。これは割り込みを表すシグナルで、主に実行中のプロセスを停止させたい時に使用するのだ。わりと頻繁に使用するので、使ったことがある人もいるかもしれない。たとえば「yes」コマンドを実行してみよう。すると端末に延々とyが表示されつづけるはずだ。その状態で[Ctrl]キー+[c]を押すことで「yes」コマンドを終了させられる(リスト32)。プログラムが応答しなくなった時や、時間がかかる処理を間違って実行してしまった時など、活用するシーンは多いので、ぜひ覚えよう。もちろん[Ctrl]+[z]でサスペンドした後に「kill」してもかまわないぞ。

他によく使われるシグナルには、プロセスを終了後再起動する「SIGHUP」やプロセスを強制終了する「SIGKILL」がある。シグナルの種類は「man」でも確認できるので一度目を通しておこう。

List 32 プログラムが応答しないときにも

```
$ yes
y
y
<…中略…>
y
y
^C
$
```

[Ctrl]+[c]キーを押してコマンドを終了

コマンドが終了してプロンプトが戻った

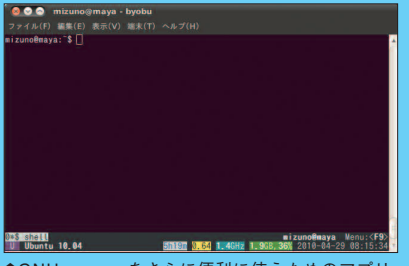
[Ctrl]+[c]キーで処理をキャンセル、と覚えておこう。

コマンドを使うことが多くなったらbyobuに乗り換えてみよう!

すでに気付いているかもしれないが、端末を閉じると実行中のコマンドは終了してしまう。「GNOME端末」であれば警告のダイアログがでるため、うっかり閉じちゃったということは少ないかもしれないが、ひょっとしたら端末がエラーで落ちてしまうようなことがあるかもしれない。それに、SSHのようなサービスを使ってサーバをリモートから管理している場合は、不意に回線が切れてしまうこともあるだろう。サーバのアップグレード中に回線が切断されてプログラムが停止するなんて、ほとんど悪夢の世界である。そこで、そんな事故を防ぎ、そしてコマンドラインをさらに便利に使うための必須ツールを紹介しよう。その名も「GNU screen」だ!

screenは仮想端末管理ソフトで、複数の端末を作成して切り替えたり、screen上で起動しているプロセスと接続している端末を分離したりできる。そしてUbuntuにはscreenをさらに便利に使う「byobu」が最初から搭載されているぞ。byobuの機能についてはいずれ改めて解説する予定だ。

Ubuntuならbyobu!!



GNU screenをさらに便利に使うためのアプリ、Ubuntu発のbyobuをぜひ使いたい。

正規表現で文字列をパターン検索

正規表現を使いこなそう！

正規表現とは、文字列のパターンを表現する方法のひとつだ。プログラミングの世界で正規表現と言った場合は、主に文字列の検索や置換を行う場合のパターンマッチングを指すことが多い。正規表現を使えば特定の文字列ではなく、任意のパターンで文字列を検索することができる。それではそんな正規表現を、シェル上から便利に使う方法を紹介しよう。

正規表現の基礎「grep」で検索しよう！

既に何度か例で紹介しているが「grep」という基本的なコマンドがある。これは与えられた正規表現にマッチする行を検索するコマンドだ。「grep」の使い方は**リスト32**の通りで、オプションの詳細は「man」コマンドで調べてみてほしい。パターンには様々な正規表現を使うことができるぞ。

指定の文字列を含む行をファイルから検索

「a」という正規表現は「a」という文字に一致する。同じように「b」という正規表現は文字「b」を表す。つまり正規表現として文字列を与えることで、その文字列を検索できるわけだ。**リスト34**は「/etc/passwd」ファイルから「root」という文字を含む行を検索する例だ。

行頭や行末の文字を指定して検索

単純に文字列だけで「grep」すると、該当の文字列が1行の中のどこにあってもマッチしてしまう。そこで文字列を検索する際のコツとして、行の頭、または末尾を起点として検索するとうまくいくことが多い。行頭を表すには「^」（英数キャレット）、行末を表すには「$」を使用する。**リスト35**は「^ad」つまり行頭がadで始まる（＝adではじまる名前の）グループを検索している例だ。行頭の指定を外してみたのが**リスト36**だ。見てのとおり、「shadow」や「lpadmin」といったグループまでマッチしてしまっている。**リスト37**は逆に「bash$」つまり行がbashで終わる（＝ログインシェルがbashの）ユーザを検索してみたぞ。

列挙した文字のどれかにマッチしたものを検索

あの文字か、この文字のどっちかにマッチするものを探したい。そういう要求はよくあるはずだ。そんなときは該当の文字を［］で囲もう。［］で囲まれた文字のうち「いずれか」にマッチするものを探すことができるぞ。**リスト38**では行頭がaかbかcで始まるユーザを検索してみた。

範囲で指定した文字や数字で検索する

文字や数字は始点と終点をハイフンでつなぐことで、範囲として指定できる。例えば前述の**リスト37**の［abc］は［a-c］と表現できるし、すべての数字にマッチさせたいなら［0-9］と書くことができる。**リスト39**では範囲を使って、10から19という範囲のグループIDを持つグループを検索してみた。

特定の文字にマッチさせない検索

特定の文字にマッチさせたくないという場合もよくあるはずだ。そのような場合は角括弧の中に「^」を書くことで列挙された文字にマッチさせないことができる。**リスト40**は範囲の前に「^」を置くことで行頭がaからv以外、つまり行頭がw、x、y、zのいずれかで始まるユーザを検索している。角括弧の中と外で「^」の働きが違うことに注意しよう。

任意の1文字にマッチさせる

「.（ピリオド）」は、任意の1文字にマッチさせることができる。文字の種類は問わないので、とにかく「なんでもいいからそこに文字があること」を指定したいときによく使うぞ。**リスト41**はまず「cut」コマンドで「/etc/passwd」からユーザ名部分のみを抽出し、パイプで「grep」に渡している。検索パターンは行頭から任意の3文字があって行末だ。つまり3文字のユーザ名のユーザを検索してる。

パターンの繰り返しを指定して検索する

「*（アスタリスク）」を使えば、直前のパターンを任意の回数繰り返すと指定することができる。**リスト42**は「行頭がsysではじまり」

List 33 grepの使い方

```
grep オプション パターン ファイル
```

■検索コマンドの基本、grepの使い方は覚えておこう。

List 34 文字列を検索

```
$ grep 'root' /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

■/etc/passwdファイルから文字列 'root' を検索する方法。

List 35 'ad'で始まる名前のグループ

```
$ grep '^ad' /etc/group
adm:x:4:mizuno
admin:x:119:mizuno
```

■'ad'ではじまる名前のグループを検索すると2つ該当。

List 36 行頭の指定を外すと……

```
$ grep 'ad' /etc/group
adm:x:4:mizuno
shadow:x:42:
lpadmin:x:105:mizuno
admin:x:119:mizuno
```

■'ad'を名前に含むグループ4つが該当した。

List 37 シェルがbashのユーザを検索

```
$ grep '/bash$' /etc/passwd
root:x:0:0:root:/root:/bin/bash
<…中略…>
mizuno:x:1000:1000:mizuno,,,:/ho
me/mizuno:/bin/bash
```

■行がbashで終わる、つまりログインシェルがbashのユーザを検索した例。

List 38 頭文字がaかbかcのユーザ

```
$ grep '^[abc]' /etc/passwd
bin:x:2:2:bin:/bin:/bin/sh
<…中略…>
avahi:x:104:111:Avahi mDNS
daemon,,,:/var/run/avahi-
daemon:/bin/false
couchdb:x:105:113:CouchDB
Administrator,,,:/var/lib/couchdb:/bin/bash
```

■［］で囲まれた文字のうち「いずれか」に該当するものを検索できる。

「その後にコロン以外の文字列が任意の回数続き」「その後にコロンがくる」つまり「syshoge」のようなユーザを検索している。さて、このリストの検索結果だが、おそらく想定通りになっていないと思う。「syslog」ユーザがマッチしているのは当然だが、「sys」ユーザがマッチしているのはおかしいと思わないだろうか？実はこれで正解なのだ。アスタリスクは直前のパターンが0回以上繰り返すことを指定している。sysユーザはsysとコロンの間に「コロン以外の文字が0文字存在する」ので、このパターンにマッチするのだ。

厳密に回数を指定したいなら、「{n,m}」という繰り返しの指定ができる。**リスト43**は**リスト42**を改造し、「sysの後にaからzのアルファベットが」「1回以上、3回まで繰り返される」文字列を検索する例だ。波括弧の前にはバックスラッシュを置く必要があるので気をつけてほしい。ちなみにmを省略すると、n回以上で上限なしという指定になるぞ。

パターンにマッチした文字列を再利用する

パターンにマッチした部分は再利用することができる。まず、再利用したい部分を（）で囲おう。この括弧も回数指定の波括弧と同様、前にバックスラッシュを置く必要がある。そして「\1、\2、\3…」がそれぞれ、パターンの最初の括弧、2番目の括弧、3番目の括弧それぞれにマッチした部分に置き換えられる。少しわかりづらいと思うので、**リスト44**の具体的な例を見てみよう。このパターンは、まず任意の数字1桁を括弧でくくっており、そして直後に\1がある。括弧でくくった任意の数字が0にマッチした時は\1も0、1の場合は\1も1になる。つまりここは00、11、22……99というパターンと同等だ。その後にコロンと任意の文字が行末まで続くので、結局このパターンは「グループIDの下2桁がゾロ目のグループ」となるわけだ。

正規表現をさらに応用してみる

「grep」と並んで最も代表的な正規表現の使用例が「sed」だろう。「sed」はストリームエディタの名前が示す通り、ファイルやパイプからの入力ストリームに対し、与えられたコマンドを適用する「非対話型エディタ」だ。

「sed」はファイルやパイプから1行ずつ文字列を読み込み、指定された処理を適用して出力する。**リスト45**を例に見てみよう。このスラッシュで囲まれた部分が正規表現アドレスだ。「sed」は正規表現「^#」がマッチする行、すなわち行頭が「#」で始まる行を検索し、その行に対して「d」コマンドを実行している。「sed」の「d」コマンドは行の削除を意味するため、結果として「#」ではじまる1行目は削除される。2行目は正規表現にマッチしないため、何も処理が適用されず、元のまま表示されている。

リスト46は単純な文字列の置換を行った例だ。「s」コマンドは置換を表し、対象となる正規表現と置換後の文字列をスラッシュで区切って指定する。「g」はグローバルの意味で、1行に複数の対象があった場合、全てに対して置換を行うことを意味している。「ubuntu.txt」から「Ubuntu」という文字列を検索し、「Kubuntu」に置き換えているのがわかるだろうか。

このように、正規表現はUnixの様々な場所で日常的に使用されている超強力なツールなのだ。

List 40 特定の文字を除外する場合

```
$ grep '^[^a-v]' /etc/passwd
www-data:x:33:33:www-
data:/var/www:/bin/sh
```

■行頭がaからv以外、つまり行頭がw、x、y、zのいずれかではじまるユーザを検索している。

List 39 範囲を指定して検索

```
$ grep ':1[0-9]:' /etc/group
uucp:x:10:
man:x:12:
proxy:x:13:
kmem:x:15:
```

■グループIDが10から19のグループを検索した例。

List 43 文字数を指定してユーザ名を検索

```
$ grep '^sys[a-z]\{1,3\}:'
/etc/passwd
syslog:x:101:103::/home/sys-
log:/bin/false
```

■「sys」の後にa-zのアルファベットが「1回以上、3回まで繰り替えされる」文字列を検索する例。

List 42 sysで始まるユーザ名を検索

```
$ grep '^sys[^:]*:' /etc/passwd
sys:x:3:3:sys:/dev:/bin/sh
syslog:x:101:103::/home/syslog:/
bin/false
```

■この指定だと、sysに続く文字の数を指定していないのでsysユーザも該当してしまう。

List 41 3文字のユーザ名を検索

```
$ cut -d : -f 1 /etc/passwd |
grep '^...$'
bin
sys
man
irc
gdm
```

■行頭の^と行末の$の間がピリオド3つ。つまり任意の3文字を指定している。

List 46 sedで文字列を置換する

```
$ sed -e 's/Ubuntu/Kubuntu/g'
ubuntu.txt
<実行結果>
# Kubuntu is an operating sys-
tem.
Kubuntu 10.04 LTS, Here for the
Long Term.
```

■Ubuntuという文字列がすべてKubuntuに置き換えられて表示された。

List 45 sedで文字列を処理する

```
$ cat ubuntu.txt
<実行結果>
# Ubuntu is an operating system.
Ubuntu 10.04 LTS, Here for the
Long Term.
$ sed -e '/^#/d' ubuntu.txt
<実行結果>
Ubuntu 10.04 LTS, Here for the
Long Term.
```

■行頭に「#」がつく1行目は無視され、2行目のみが表示される。

List 44 IDの下2桁がゾロ目のグループ

```
$ grep '\([0-9]\)\1:.*$'
/etc/group
voice:x:22:
www-data:x:33:
video:x:44:mizuno
<…以下略…>
```

■パターンにマッチした部分を再利用した検索も可能なのだ。