

[改訂新版]

シェルスクリプト 基本リファレンス

`#!/bin/sh` で、ここまでできる

山森丈範 [著]

技術評論社

本書について——改訂にあたって

『シェルスクリプト基本リファレンス』の初版が発行されたのは2005年のことでした。OSに依存せず、LinuxでもFreeBSDでもSolarisでもそのまま動作するシェルスクリプトとはいえ、細かい点を見るとOSやシェルの種類によって使える文法、コマンド、オプション等に差異があります。そのような差異を明らかにしつつ、初心者にもわかりやすい解説を行って1冊の本にまとめたのが本書でした。シェルスクリプトを書く際には、筆者自身も本書を参照して各OSの対応状況をチェックすることがあり、それほどに本書が役に立つ内容となったのではと自負しています。

元々バージョン更新による変化が少ないシェルスクリプトですが、初版から少々年月が経った今、本書の内容を更新するべき箇所も増えてきました。たとえば、FreeBSD 4.xでは/bin/shの組み込みコマンドだったprintfが現在では外部コマンドに変わっている、bash3以降で使えるようになった{1..10}などの連番のブレース展開などです。また、bashを#!/bin/shのスクリプトで起動した場合は動作しないため初版では触れなかったプロセス置換について、改訂版では取り上げることにしました。そのほか、本書に掲載されたすべてのシェルスクリプトについて、Linux/FreeBSD/Solarisで再度動作確認を行い、本文解説等の文章の一部については、よりわかりやすいように表現を修正しました。

これからシェルスクリプトを学ぶ方はもちろん、中級者や上級者の方にとっても本書が「いつも手元に置いておきたい本」になれば幸いです。




2011年3月

山森 丈範

本書の特徴

プログラマではなく、普通に道具としてコンピュータを使用している人でも、たとえば、カレントディレクトリの複数のファイルそれぞれに一定の処理を施したいとか、所定のオプションを付けて一連のコマンドを起動したいとかいった状況はよく発生します。こんなとき、シェルスクリプトを知っていれば、これらの処理を行うツールがすぐに簡単に作れます。

シェルスクリプトはプログラム言語の一種ですが、コンパイルなどの操作が不要で、誰でも気軽に書いてしまう言語です。プログラム言語の代表格であるC言語は、ある意味プログラマだけが知っていればいい言語ですが、シェルスクリプトは、プログラマ以外のユーザも含めたごく普通のユーザが、自分の作業の効率アップのために日常的に使用する言語なのです。

本書では、おもにLinuxを中心に標準シェルとして使用されているbashを取り上げ、このbashを使ったシェルスクリプトの書き方をひととおり解説しています。執筆にあたっては、bashを基本としながらも、bash以外のBシェル(Bourne Shell)系のシェルとの互換性・移植性を重視し、bash独自と思われる拡張部分よりも、基本的なBourne Shellとしての解説に重点を置きました。このため、すべての項目について、bashのほかに、FreeBSDのsh(/bin/sh)とSolarisのsh(/bin/sh)で動作確認を行い、対応状況をそれぞれ    のようにアイコンを用いて表示してあります。この表示は、シェルスクリプトの互換性・移植性のための大きな参考となるでしょう。また、これら3種類のシェルすべてに○印のアイコンが表示されている項目のみを使用すれば、OSに依存しない、移植性の高いシェルスクリプトを記述することができますはずです。

シェルスクリプトをまったく書いたことがない人は、本書を読み、まずは「Hello World」から書き始めていただきたいです。また、ある程度のシェルスクリプトを書いたことがある人には、本書の記述をヒントに、シェルスクリプトの小技・裏技を再発見していただければ幸いです。

※初版『シェルスクリプト基本リファレンス』より。

本書が想定している読者

本書では、ユーザとして、コマンドラインでのLinux/UNIXの基本操作ができる人を想定しています。具体的には、ls、cp、rmなどの基本的なコマンドが使えれば本書の内容が理解できるでしょう。

また、本書の中には、シェルスクリプトとC言語とを比較している部分がありますが、C言語については必ずしも理解している必要はありません。これはあくまでさらに理解を深めるための参考事項にすぎないため、適宜読み飛ばしても差し支えないものです。

本書が対象としている環境

本書の記述内容を実行するには、bashが動作する環境が必要です。

bashは、LinuxやMac OS Xでは、標準シェルとしてインストールされているため、これらのユーザは、とくに設定変更を行っていないかぎり、すでにbashを使用しているはずです。FreeBSDやSolarisでは、bashはOS標準ではありませんが、ソースファイルやバイナリパッケージなどからbashをインストールして使うことが可能です。

また、仮にbashがなくても、sh、ash、kshなどのBシェル系のシェルがあれば、前述のとおり本書の    のすべてに○がついている項目については、Bシェル系として共通に動作するでしょう。

なお、本書ではOSとしてLinuxなどのUNIX系OSを想定していますが、bashが動作する環境でさえあれば、たとえばWindowsのCygwin環境のような、非UNIX系環境であってもかまいません。

動作確認の環境について

動作確認時のOSのバージョンとシェルはそれぞれ、Fedora 13(Linux)のbash 4.1.7およびbash 3.2.33、FreeBSD 8.1の/bin/sh、Solaris 10の/bin/shです。これら以降のバージョンを使用していれば問題なく動作すると思われます。また、これら以前のバージョンであっても、ほとんどが問題なく動作すると思われます。

ただし、上記と異なるバージョン、環境においては、本書での説明内容と動作等が異なる場合もあります。その点については、あらかじめご承知おきください。

本書の記述について

項目(タイトルデザイン)

① 概論やサンプルスクリプトを扱った項目と② リファレンスとしての使用を想定とする項目とで、異なるタイトルデザインを採用しました。とくに②ではタイトル部分に項目名とそのポイントを併記していますので、一目で内容を確認できるようになっています。

対応するOS/シェルを表すアイコン

本書の項目タイトルでは、動作確認をした環境であるLinux(bash)、FreeBSD(sh)、Solaris(sh)の対応状況をアイコンで表示しています。

また、解説内で、とくに制限のある方法に関する記述部分については、以下の例のように **Warning** として対応状況を別途アイコンで表示した上で、グレー地になっています。

例

①

シェル文法における複合コマンド

②

if文

条件判断によってプログラムを分岐する



例

Linux
(bash)

対応

FreeBSD
(sh)

制限あり

Solaris
(sh)

非対応

例

パターンを()で囲むこともできる

bashまたはFreeBSDの場合は、case文のパターンの両端を()で囲んで、リストFのように記述することもできます。ただし、このように記述してしまうと従来のshとの互換性がなくなり、またサードパーティのshとも動作しなくなるため、この記述法は用いないほうがよいでしょう。

例としてパターンを()で囲んだ例
case 'uname -s' in
(Linux)
echo "GNU/Linuxです"
*)
*)
*)



Warning

Linux (bash) FreeBSD (sh) Solaris (sh)

この方法には制限があります。

書式 中で使用している記号

①

~

例

ディレクトリ名

実際のユーザ入力で置き換えられる部分は、~ のように表示しています。上の例の **ディレクトリ名** では、実際にシェルスクリプトを記述する際は、ディレクトリ名(たとえば/some/dir)で置き換えて使用するという意味です。

また、この記号は適宜 **基本事項** の説明でも **書式** の説明と対応する個所に用いています。

②

[]

例

ディレクトリ名

[および] で囲まれた部分は、省略できます。上の例では **ディレクトリ名** は必要に応じて指定してもよいし、省略してもよいという意味です。

③

|

例

変数名 関数名

| で区切られた要素は選択して指定できます。上の例は **変数名** または **関数名** という意味です。

④

...

例

引数 ...

直後に... が続く要素は、繰り返してもよいという意味です。

上の例は、... も [] の中に入っていますので、引数を任意に0個以上付けることができるという意味になります。

本書について	iii
本書の特徴	iv
本書の記述について	vi
目次	viii

第0章 シェル&シェルスクリプトの基礎知識 1

0.1 概要	2
シェルとシェルスクリプト	2
Column シェルスクリプトが使用されている場面	4
0.2 いろいろなシェル	5
基本はBourne Shell	5
Column 自分のログインシェルを調べる	7
0.3 プログラムであるシェルスクリプト	9
シェルスクリプトとC言語との比較	9
0.4 移植性の高いシェルスクリプトについて	12
シェルスクリプトの移植性	12

第1章 シェルスクリプト入門 13

1.1 Hello World	14
シェルスクリプト作成の流れ ◆Hello Worldを書いてみよう	14
1.2 #!/bin/shの意味	16
#!/bin/shの意味 ◆シェルスクリプトの1行目	16
1.3 実行方法について	18
いろいろな実行方法 ◆コマンドとして実行する以外には	18

第2章 シェルスクリプトの基本事項 19

2.1 シェルスクリプトはフリーフォーマット	20
シェルスクリプトはフリーフォーマット ◆改行位置やインデントなどはかなり自由	20
2.2 コメントの書き方	23
コメントの書き方 ◆コメントを記入するには#記号を使う	23
2.3 コマンドの終了ステータス	25
コマンドの終了ステータス ◆条件判断を行う際に利用される	25

第3章 シェル文法の循環構造 27

3.1 コマンド→パイプライン→リストの循環	28
------------------------	----

コマンド→パイプライン→リストの循環	28
3.2 コマンド／パイプライン／リスト	29
単純コマンド ◆一般のUNIXコマンドは単純コマンドとして実行できる	29
複合コマンド ◆構文／サブシェル／シェル関数などは複合コマンドとして解釈される	30
コマンド ◆単純コマンドと複合コマンドとを合わせてコマンドと呼ぶ	31
Column 「いわゆる」UNIXコマンド(単純コマンド)の調べ方	31
パイプライン ◆コマンドの標準出力を別のコマンドの標準入力に接続する	32
リスト ◆パイプラインを改行などで区切って並べたもの	35
3.3 &&リスト／ リスト	37
&&リスト ◆簡単な条件判断を行える	37
リスト ◆簡単な条件判断を行える	39

第4章 複合コマンド 41

4.1 概要	42
シェル文法における複合コマンド	42
4.2 構文	43
if文 ◆条件判断によってプログラムを分岐する	43
case文 ◆文字列をパターンごとに場合分けしてプログラムを分岐する	49
for文 ◆変数に、指定の値をそれぞれ代入しながらループする	55
算術式のfor文 ◆ループ変数を使い算術式を評価しながらループを繰り返す	61
while(until)文	
◆条件が真であるかぎり(偽になるまで)ループを繰り返すにはwhile文を使う	63
select文 ◆選択メニューを表示しユーザ入力を受け付ける	69
4.3 サブシェルとグループコマンド	72
サブシェル ◆リストをまとめて別のシェルで実行する	72
グループコマンド ◆リストを1つのコマンドとしてまとめる	74
4.4 シェル関数	76
シェル関数 ◆一定の処理を関数としてまとめる	76
4.5 算術式の評価と条件式の評価	80
算術式の評価 (()) ◆算術演算を行いその結果によって終了ステータスを返す	80
条件式の評価 [[]] ◆条件式を評価し、その結果によって終了ステータスを返す	83

第5章 組み込みコマンド 85

5.1 概要	86
基本の組み込みコマンドについて	86
5.2 組み込みコマンド(基本)	87
:コマンド ◆何もしないで単に「0」の終了ステータスを返す	87
.コマンド ◆現在実行中のシェルに別のシェルスクリプトを読み込ませる	90
break ◆for文/while文のループを途中で抜ける	93

continue	◆for文/while文のループを次の回に進める	94
cd	◆別のディレクトリに移動する	95
eval	◆引数を再度解釈しコマンドを実行する	98
exec	◆新しいプロセスを作らずにコマンドを起動する	100
exit	◆シェルスクリプトを終了する	103
export	◆シェル変数を環境変数としてエクスポートする	105
getopts	◆シェルスクリプトの引数に付けられたオプションを解析する	108
read	◆標準入力からの入力をシェル変数に読み込む	111
readonly	◆シェル変数を読み込み専用にする	115
return	◆シェル関数を終了する	118
set	◆シェルにオプションフラグをセットする、または位置パラメータをセットする	120
shift	◆位置パラメータをシフトする	123
trap	◆シグナルを受け取った時に指定のコマンドを実行させる	125
type	◆外部コマンドのフルパスを調べたり、組み込みコマンドかどうかをチェックしたりする	127
umask	◆シェル自体のumask値を設定/表示する	128
unset	◆シェル変数またはシェル関数を削除する	131
wait	◆バックグラウンドで起動したコマンドの終了を待つ	133

第6章 組み込みコマンド 2 135

6.1 概要	136
外部コマンド版も存在する組み込みコマンドと、 拡張された組み込みコマンドについて	136
6.2 組み込みコマンド(外部コマンド版もあり)	137
echo ◆任意のメッセージを標準出力に出力する	137
false ◆単に「1」の終了ステータスを返す	140
kill ◆プロセスにシグナルを送る	141
printf ◆メッセージを一定の書式に整形して標準出力に出力する	143
pwd ◆カレントディレクトリの絶対パスを表示する	145
test ◆シェルスクリプトにおいて各種条件判断を行う	147
true ◆単に「0」の終了ステータスを返す	150
6.3 組み込みコマンド(拡張)	151
builtin ◆シェル関数と同名の組み込みコマンドを優先的に実行する	151
command ◆シェル関数と同名の組み込みコマンドまたは外部コマンドを優先的に実行する	152
let ◆算術式を評価するのにletコマンドを使う方法もある	153
local ◆シェル関数内でローカル変数を使う	154
6.4 組み込みコマンド(その他)	156
その他の組み込みコマンド	156

第7章 パラメータ 157

7.1 概要	158
シェルにおけるパラメータ	158

7.2 シェル変数の代入と参照	159
シェル変数の代入と参照 ◆変数や定数を使うにはシェル変数を使う	159
7.3 位置パラメータ	162
位置パラメータ ◆シェルスクリプトやシェル関数の引数を参照する	162
7.4 特殊パラメータ	165
特殊パラメータ \$0 ◆起動されたシェルスクリプト名(第0引数)を参照する	165
特殊パラメータ "\$@"	
◆シェルスクリプトやシェル関数の引数すべてをそのまま引き継ぐ	167
特殊パラメータ \$*	
◆シェルスクリプトやシェル関数の引数すべてを1つに連結して参照する	169
特殊パラメータ \$# ◆シェルスクリプトやシェル関数の引数の個数を参照する	171
特殊パラメータ \$? ◆終了ステータスを参照する	173
特殊パラメータ \$!	
◆最も新しくバックグラウンドで起動したコマンドのプロセスIDを参照する	175
特殊パラメータ \$\$ ◆シェル自身のプロセスIDを参照する	176
特殊パラメータ \$_ ◆現在のシェルに設定されているオプションフラグを参照する	177
特殊パラメータ \$_ ◆直前に実行したコマンドの最後の引数を参照する	178
7.5 環境変数	179
環境変数の設定 ◆シェル変数をexportして環境変数を設定する	179
環境変数の一時変更 ◆単純コマンドの左側に環境変数の代入文を記述する	180
7.6 特別な意味を持つシェル変数	181
PATH ◆外部コマンドの検索パスを設定するシェル変数	181
PS1 / PS2 ◆シェルのプロンプトを設定するシェル変数	182
HOME ◆自分自身のホームディレクトリが設定されているシェル変数	184
IFS ◆単語分割に用いられる区切り文字が設定されているシェル変数	184

第8章 パラメータ展開 185

8.1 概要	186
パラメータ展開の概要	186
Column パラメータ展開とダブルクォート	186
8.2 条件判断をともなうパラメータ展開	187
\${パラメータ:-値}と\${パラメータ-値} ◆パラメータのデフォルト値を指定する	187
Column 空文字列のパラメータをセットするには	188
\${パラメータ:=値}と\${パラメータ=値} ◆パラメータにデフォルト値を代入する	189
\${パラメータ:?値}と\${パラメータ?値}	
◆パラメータ未設定時にエラーメッセージを出してシェルスクリプトを終了する	191
\${パラメータ:+値}と\${パラメータ+値}	
◆パラメータが設定されている場合のみ指定の値に展開する	193
\${#パラメータ} ◆パラメータの値の文字列の長さを求める	195

<code>\${パラメータ#パターン}</code> と <code>\${パラメータ##パターン}</code>	197
◆パラメータの値の文字列の左側から一定のパターンを取り除く	
<code>\${パラメータ%パターン}</code> と <code>\${パラメータ%%パターン}</code>	199
◆パラメータの値の文字列の右側から一定のパターンを取り除く	
<code>\${パラメータ:オフセット}</code> と <code>\${パラメータ:オフセット:長さ}</code>	201
◆オフセットや長さを指定してパラメータの値の文字列を切り出す	
<code>\${パラメータ/パターン/置換文字列}</code> と <code>\${パラメータ//パターン/置換文字列}</code>	202
◆パターンを指定してパラメータの値の文字列を置換する	
<code>\${!変数名@}</code> または <code>\${!変数名*}</code>	203
◆指定した文字列で始まる変数名を一覧表示する	
間接参照 <code>\${!パラメータ}</code>	204
◆パラメータの値をパラメータ名とみなし、さらにその値を参照する	

第9章 クォートとコマンド置換 205

9.1 概要.....	206
クォートとコマンド置換.....	206
9.2 クォート.....	207
シングルクォート ' ' ◆文字の特殊な意味を打ち消して文字列を使用する.....	207
ダブルクォート " " ◆パラメータ展開とコマンド置換を除いて文字の特殊な意味を打ち消す.....	209
バックスラッシュ \ ◆次の1文字の特殊な意味を打ち消す.....	211
9.3 コマンド置換.....	213
コマンド置換 ` ` ◆コマンドの引数またはコマンド名を別のコマンドの標準出力で置換する.....	213
コマンド置換 \$() ◆コマンドの引数またはコマンド名を別のコマンドの標準出力で置換する.....	216

第10章 各種展開 219

10.1 概要.....	220
シェルにおける各種展開.....	220
10.2 パス名展開.....	221
パス名展開 * ◆0文字以上の任意の文字列にマッチさせる.....	221
パス名展開 ? ◆任意の1文字にマッチさせる.....	223
パス名展開 [a-z] ◆指定した条件の1文字にマッチさせる.....	225
10.3 ブレース展開.....	229
ブレース展開 {a,b,c} ◆複数の文字列の組み合わせから文字列を生成する.....	229
10.4 算術式展開.....	232
算術式展開 \$(()) ◆算術式を評価しその演算結果の数値に展開する.....	232
10.5 チルダ展開.....	233
チルダ展開 ~ ◆ユーザのホームディレクトリに展開する.....	233

10.6 プロセス置換.....	234
プロセス置換 <() / >() ◆FIFOに接続した別プロセスを起動し、そのFIFO名に置換する.....	234
10.7 単語分割.....	237
単語分割 (IFS) ◆パラメータ展開などの結果をコマンド名や引数に分割する.....	237

第11章 リダイレクト 239

11.1 概要.....	240
リダイレクト.....	240
11.2 いろいろなリダイレクト.....	241
標準入力のリダイレクト < ◆コマンドの標準入力にファイルを入力する.....	241
Column ファイル記述子について.....	242
標準出力のリダイレクト > ◆コマンドの標準出力をファイルに出力する.....	243
標準出力のアPENDモードでのリダイレクト >> ◆コマンドの標準出力をファイルに追加出力する.....	245
標準エラー出力のリダイレクト 2> ◆コマンドの標準エラー出力をファイルに出力する.....	246
ファイル記述子を使ったリダイレクト >& ◆オープン済みの標準出力や標準エラー出力などを複製する.....	248
標準出力と標準エラー出力の同時リダイレクト &> ◆bashで標準出力と標準エラー出力を同時にファイルにリダイレクトするには&>が使える.....	251
ファイル記述子のクローズ >&- / <&- ◆ファイル記述子をクローズするには>&-や<&-を用いる.....	252
読み書き両用オープン <> ◆コマンドのファイル記述子を読み書き両用でオープンしたファイルにリダイレクトする.....	253
ヒアドキュメント ◆コマンドの標準入力に一定の文書を入力する.....	254
ヒアストリング ◆コマンドの標準入力に一定の文字列を入力する.....	256
Column ヒアドキュメントとヒアストリングの実際.....	258

第12章 よく使う外部コマンド 259

12.1 概要.....	260
シェルスクリプトで使う外部コマンド.....	260
12.2 シェルスクリプトならではのコマンド.....	261
expr ◆シェルスクリプトで数値計算を行う.....	261
basename ◆ファイル名からディレクトリ名部分や拡張子を取り除く.....	263
dirname ◆ファイル名からそのディレクトリ名部分のみを取り出す.....	265
cat ◆ファイルを連結し標準出力に出力する.....	267
sleep ◆一定時間待つ.....	268
12.3 一般コマンド.....	269
一般コマンド ◆よく使う一般コマンド.....	269

grep	◆ファイルから特定の文字列を含む行を抜き出す	270
wc	◆ファイルの行数／単語数／ファイルサイズを表示する	270
head	◆ファイルの先頭から指定の行数だけを取り出す	271
tail	◆ファイルの末尾から指定の行数だけを取り出す	271
touch	◆ファイルの更新時刻を更新する	272
sed	◆ファイルの中の文字列を置換する	272

第13章 配列 273

13.1	概要	274
	シェル上での配列	274
13.2	配列への代入と参照	275
	配列への代入と参照 ◆bashではシェル変数として配列も扱える	275
13.3	配列の一括代入と一括参照	277
	配列の一括代入と一括参照	
	◆配列のすべての要素について値を一括代入したり一括参照したりできる	277
13.4	bash以外のシェルで配列を使う方法	279
	bash以外のシェルで配列を使う方法	
	◆bash以外のシェルでもevalコマンドを用いて配列と同様の処理ができる	279

第14章 シェルスクリプトのノウハウ&定石 281

	クォートのネ스팅パズル	282
	if文の代わりに&&や を使う	284
	do~while構造の実現	285
	すべての引数についてループする	287
	一定回数ループする	290
	ラッパースクリプト	293
	ファイル名に日付文字列を含ませる	294
	ファイルを1行ずつ読んでループする	295

Appendix サンプルスクリプト 299

	引数の解釈状況をチェックする	300
	標準出力／標準エラー出力の出力先をチェックする	302
	.tar.gz/.tar.bz2自動展開	304
	Shift_JIS→EUC-JP一括変換	306
	Column 文字コードの変換を行うフィルタコマンド	307
	EUC-JPの文字一覧出力	308
	プログレスバー	311

索引	314
----	-----

>第0章

シェル&シェルスクリプトの 基礎知識

0.1	概要	2
0.2	いろいろなシェル	5
0.3	プログラムであるシェルスクリプト	9
0.4	移植性の高いシェルスクリプトについて	12

シェルとシェルスクリプト

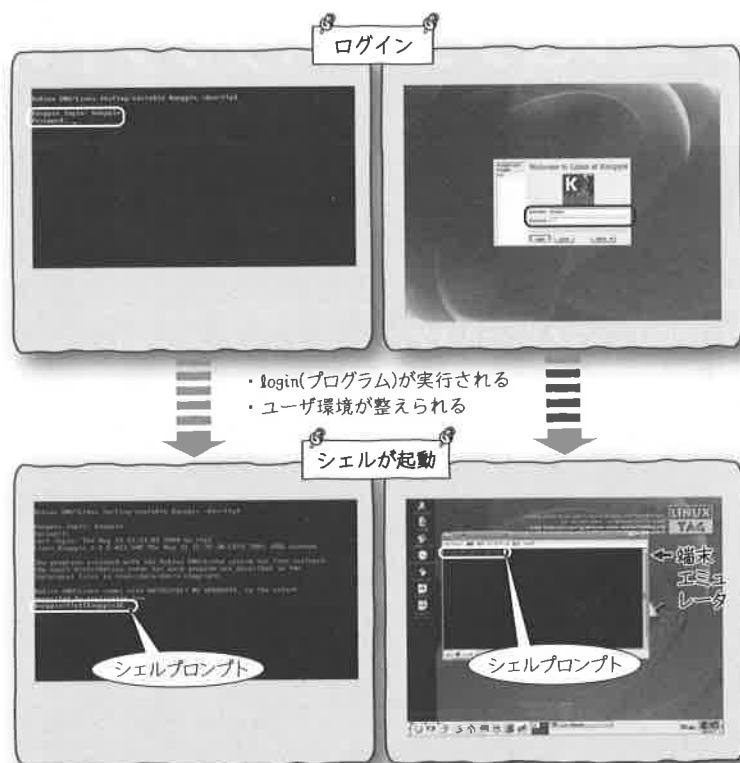
シェルは最初のプログラム

シェル(shell)は、LinuxなどのUNIX系OSにログインして最初に動作するプログラムです。ユーザが実行する各種プログラムは、基本的にはこのシェルを通して起動されます。このように、ログイン直後に起動するシェルのことを**ログインシェル**と呼びます(図A左)。

もっとも、LinuxなどのOSにグラフィカルモードでログインした場合は、すぐにはシェルが動作していないかもしれません。しかし、この場合も、xtermやkterm、またはKDEのKonsoleなどの端末エミュレータ(*terminal emulator*)と呼ばれるウィンドウを開けば、その中にシェルが起動するはずです(図A右)。

シェルは画面に**プロンプト(prompt)**を出して、ユーザにコマンド入力进行を促します。プロンプトは、bashなどのBシェル系のシェル(詳しくは後述)であればデフォルトで「\$」ですが、設定により「**ユーザ名@ホスト名[ディレクトリ名]\$**」のような格好をしているかもしれません。ユーザはこのプロンプトに対して、実行したいコマンドを入力します。

図A ログインしてシェルが起動したところ(左: テキストモード、右: グラフィカルモード)



シェルスクリプトはコマンドをファイルに記述したもの

このように、コマンド入力は通常はキーボードから行われます。しかし、いつも同じようなコマンドを毎回入力するのは大変です。そこで、実行すべきコマンドをあらかじめファイルに書いておき、そのファイルをシェルに読み込ませて実行することができます。これが**シェルスクリプト(shell script)**です。

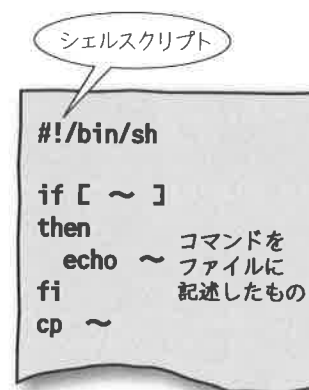
シェルスクリプトは、コマンドをキーボードから入力する代わりにファイルに記述したものにすぎません。ただし、シェルスクリプトでは、単純にコマンドを実行するだけでなく、for文やif文などの構文によってループや条件判断を行ったり、シェル変数に値を代入したり参照したりすることができるため、シェルスクリプトを使ってかなり手の込んだプログラムを記述することが可能です。

なお、キーボードからの入力時には、lsコマンドやcpコマンドなどの単純コマンドを実行する場合がありますが、キーボードからの入力でも、for文やif文などの構文を直接入力することや、シェル変数を操作することが可能です。

シェルの2つの側面

このように、シェルには、ユーザのコマンド入力用のユーザインターフェース(コマンドインタープリタ)としての側面と、シェルスクリプトの内容を文法的に解釈しながら逐次プログラムを実行するインタプリタプログラムとしての2つの側面があるといえます。

現在、bashなどの高機能シェルは、この両面ともに強化されており、コマンド入力用にもシェルスクリプト記述用にも問題なく使用できます。



Column

シェルスクリプトが使用されている場面

シェルスクリプトは、次のようなところでも使われています。

起動スクリプト(rcスクリプト)

LinuxなどのUNIX系OSの起動時には、起動スクリプト(rcスクリプト)と呼ばれるシェルスクリプトが実行されます。起動スクリプトでは、OS自体の初期設定や、各種サーバプロセスを起動するためのコマンドが実行されます。起動スクリプトの具体的なファイルの配置はOSによって異なりますが、LinuxでもFreeBSDでもSolarisでも、ls /etc/rc*とコマンド入力すれば、多数の起動スクリプトが表示されるはずです(図a)。

一般コマンド

また、/usr/binなどに配置されているシステム標準のコマンドの中には、実行バイナリファイルではなく、シェルスクリプトとしてインストールされているものが多数存在します。具体的にどのコマンドがシェルスクリプトなのかはOSによって異なりますが、file /usr/bin/*とコマンド入力して「Bourne shell script text executable」のように表示されるコマンドがシェルスクリプトです。シェルスクリプトは結局は単なるテキストファイルなので、そのテキストを表示してみれば、そのコマンドが内部でどのようなコマンドを実行しているのかを知ることができるでしょう。

図a ls /etc/rc*の実行例(これらはすべてシェルスクリプト)

```
$ ls /etc/rc*
/etc/rc /etc/rc.local /etc/rc.sysinit

/etc/rc.d:
init.d rc.local rc0.d rc2.d rc4.d rc6.d
rc rc.sysinit rc1.d rc3.d rc5.d

<中略>
/etc/rc3.d:
K01smartd K35smb K87irqbalance S10network
K02NetworkManager K36lisa K87mcstrans S11auditd
K02NetworkManagerDispatcher K50netconsole K87multipathd S12syslog
K02avahi-daemon K60crond K87restorecond S13named
K02avahi-dnssconfd K60vboxdrv K88nasd S13rpcbind
K02dhcdd K60vboxnet K88pcscd S14nfslock
K02haldaemon K65oidentd K88wpa_supplicant S26udev-post
K03yum-updatesd K68rpcidmapd K89dund S26ypserv
K05anacron K69rpcgssd K89hidd S27ypbind
K05atd K69rpcsvcgssd K89netplugd S28autofs
K05sasauthd K73ldap K89pand S56xinetd
K10ConsoleKit K73winbind K89rdisc S60nfs
K10 cups K74lm_sensors K90bluetooth S65dhcpcd
K10psacct K74nscd K92ip6tables S85gpm
K16rarpd K74ntpd K92iptables S90FreeWnn

<以下略>
```

基本はBourne Shell

Bシェル系とCシェル系

UNIX系OSで使用されているシェルには、bashだけでなく、複数の種類のシェルが存在し、これらは表Aのように、Bシェル系とCシェル系の2種類に分類できます。

本来のUNIXの標準シェルはshです。shはBourne Shellと呼ばれ、shから派生したシェルがBシェル系です。このshこそが、シェルスクリプトを記述するための標準シェルです。

しかし、shは、ユーザのコマンド入力用のシェルとしては低機能です。そこで、ヒストリ機能、エイリアス機能、ジョブコントロール機能などの機能を備え、かつ、シェル文法をC言語風に改変した新しいシェルとしてcshが開発されました。このcshから派生したシェルがCシェル系です。

cshは、UNIXの新しい標準シェルとして受け入れられ、多くのユーザがcshをログインシェルとして使いました。しかし、cshの文法はshとは異なっており、cshでシェルスクリプトを記述するには多くの問題が存在します。たとえば、標準エラー出力のみのリダイレクトができないとか、シェル関数が使えないとかです。このためcshユーザは、コマンド入力用のシェルとしてはcshを使いながら、シェルスクリプト記述時にはshを使うというように、2つのシェルの頭を切り替えながら使う必要がありました。

bashの登場によるBシェル系の復権

その後、あくまでBシェル系として、shの上位互換性を保ちながら、cshのヒストリ機能、エイリアス機能、ジョブコントロール機能などを取り入れ、さらに、タブによるファイル名やコマンド名の補完機能や、コマンドライン編集機能を備えたbash(Bourne-Again Shell)が登場しました。bashはコマンド入力用のシェルとしても、シェルスクリプト記述用のシェルとしても問題なく使用できます。bashの登場により、「Bourne-Again Shell」の名前のごとくBシェル系が復権し、以降bashのユーザが増えていきました。

一方、Cシェル系のほうでも、cshの機能拡張版として、タブによる補完機能や、コマンドライン編集機能を備えたtcshが登場し、以降、あくまでCシェル系を使いたいというユーザはtcshを使うようになりました。

現在ではLinuxが普及し、bashがLinuxの標準シェルになっていることから、多くのユーザがbashをログインシェルとして使用しています。また、Mac OS Xでは、v10.3(Panther)以降、標準シェルが以前のtcshからbashに変更されたため、ますますbashユーザが増えていくものと思われます。bashがログインシェルであれば、コマンドライン上のシェル文法とシェルスクリプト上のシェル文法が一致し、かつコマンド入力時には補完機能などの便利な機能が使えるという利点があります。

表A おもなシェルの分類

	低機能	⇒	高機能
Bシェル系	sh (Bourne Shell) (ash)	⇒	bash (Bourne-Again Shell) (zsh) (ksh)
Cシェル系	csh	⇒	tcsh

なお、Linuxの多くのディストリビューションでは、shはbashへのシンボリックリンクになっており、/bin/shを実行しても実際にはbashが起動します。また、bashはLinuxだけでなく、FreeBSDやSolarisなどのOSにもインストールして使用することが可能です。

ほかのシェルについて

● ash

bashは従来のshの上位互換ですが、従来のshにはない多くの機能や文法が拡張されています。一方、これらの拡張を行わず、なるべく従来のshに近い実装を行ったシェルがashです。ashはBSD系UNIXを由来とするシェルです。UbuntuなどのDebian系Linuxでは、ashはdashという名前前でインストールされており、/bin/shはdashへのシンボリックリンクになっています。

● ksh

Solarisのshは、拡張機能のない従来のshです。そこで、各種拡張機能を追加したシェルであるksh(Korn Shell)が、Solarisに標準で付属しています。Solarisの/usr/binなどの標準コマンドの中にはkshスクリプトとして記述されているものがあります。もちろんkshもBシェル系のシェルです。

● zsh

zshは、Bシェル系(おもにksh)とCシェル系(tcsh)の両方の特長を取り入れ、さらに大幅に機能を拡張したシェルです。zshはBシェル系に分類されますが、zshのデフォルトのプロンプトはcsh系と同じく「%」になります。また、シェル変数の展開時の単語分割の仕様などが、shやbash等とは異なっています。

シェルのプロンプトについて

Bシェル系の標準プロンプトは「\$」、Cシェル系の標準プロンプトは「%」です。文献などでコマンドラインの説明をする際には、これらのプロンプトを使ってどちらのシェル上でのコマンドかを区別することがあります。

B シェル系の場合(例)

```
$ LANG=C; export LANG
```

C シェル系の場合(例)

```
% setenv LANG C
```

Column

自分のログインシェルを調べる

各ユーザのログイン時には、環境変数(およびシェル変数)SHELLにログインシェルの絶対パスが設定されます。したがって図a①②のようにSHELLの値を表示すれば、自分のログインシェルがわかります。

また、fingerコマンドを使ってログインシェルを調べることもできます。図bのように「ユーザ名」を引数に付けてfingerコマンドを実行すると、そのユーザの情報の一部としてログインシェルが表示されます。

シェルのバージョンを調べる

シェルのバージョンを調べる方法は、シェルの種類によって違います。詳しくはそれぞれのシェルのオンラインマニュアルを参照してください。

bashの場合は、図cのように、シェル変数BASH_VERSIONの値を表示するか、またはbash自体の起動時に--versionオプションを付けることによって確認できます。

また、tcshの場合は、図dのようにシェル変数versionの値によって確認できます。

図a 自分のログインシェルを調べる

\$ echo \$SHELL	①シェル変数SHELLの値を表示
/bin/bash	bashであることがわかる
\$ printenv SHELL	②環境変数SHELLの値を表示
/bin/bash	同様にbashであることがわかる

図b fingerコマンドでログインシェルを調べる

\$ finger guest	ユーザ「guest」のログインシェルを調べる
Login: guest	Name: Guest User
Directory: /home/guest	Shell: /bin/bash ログインシェルはbash
No mail.	
No Plan.	

図c bashのバージョンを調べる

```
$ echo $BASH_VERSION  シェル変数BASH_VERSIONの値を表示する
2.05b.0(1)-release    バージョンが表示される
$ bash --version       bashの起動時に--versionオプションを付けてもよい
GNU bash, version 2.05b.0(1)-release (i386-redhat-linux-gnu)
                        バージョンが表示される
Copyright (C) 2002 Free Software Foundation, Inc.
```

図d tcshのバージョンを調べる

```
$ tcsh                  tcshを起動する
% echo $version
tcsh 6.12.00 (Astron) 2002-07-23 (i386-intel-linux) options
8b,nls,dl,a l,kan,rh,color,dspm,filec バージョンが表示される
% exit                  tcshを終了する
```

ログインシェルを変更する

ログインシェルを、現在とは別のシェルに変更するには、chshコマンドを使います。たとえば、ログインシェルをbashに変更するには、図eのようにシェルの絶対パスを指定してコマンド入力します。なお、変更先のシェルは、`/etc/shells`というシェルの一覧を記述したファイルに含まれている必要があります。詳しくはchshコマンドのオンラインマニュアルを参照してください。

図e ログインシェルをbashに変更する

```
% chsh -s /bin/bash      chshコマンドに-sを付けて/bin/bashを指定
Password:                パスワードを入力
```

シェルスクリプトとC言語との比較

シェルスクリプトはプログラムの一種です。シェルスクリプトを使ってかなり複雑なプログラムを記述することも可能です。しかし、一般にはプログラムといえはまずC言語を思い浮かべることが多いでしょう。実際に、UNIX系OSの本体やアプリケーションの多くはC言語で記述されています。そこで、シェルスクリプトとC言語との違いを考えてみることにしましょう。

シェルスクリプトとC言語との違いをまとめると表Aのようになります。大きな違いは、シェルスクリプトがインタラクティブ(対話形式)言語であり、コンパイルが不要で、記述したシェルスクリプトを逐次シェルが解釈しながら実行するという点です。

シェルスクリプトの向き／不向き

シェルスクリプトは、C言語とは違って、実行ファイルがCPUが直接実行できるバイナリファイルになっていないため、実行速度は遅くなります。しかし、Linux上で記述したシェルスクリプトを、FreeBSDやSolarisへ持ち込んでも、そのまま動作が可能です。また、シェルスクリプトには、プログラムの修正が容易、プログラムの記述時に細かいエラー処理などを簡略化できる、などの利点があり、結果的にプログラムを早く作成できます。プログラム中で文字列を扱う場合、C言語では文字列用の配列を宣言したり、文字列操作のライブラリ関数を呼び出したりと、その扱いが面倒ですが、シェルスクリプトでは、シェル変数や文字列を単に並べるだけで文字列の連結ができるなど、扱いが容易です。

シェルスクリプトは、基本的にはすでに存在する外部コマンドを呼び出しながら処理を進めます。したがって、シェルスクリプトは既存のコマンドを組み合わせで一定の動作を行わせるのに適しています。一方、C言語は、既存のコマンドの組み合わせではできない、新たなアプリケーションなどを記述するのに適しています。

表A シェルスクリプトとC言語との比較

シェルスクリプト	C言語
インタラクティブ言語	コンパイル言語
コンパイルは不要ソースファイルと実行ファイルが同一	あらかじめコンパイルが必要ソースファイルと実行ファイルが別
実行ファイルはテキストファイル	実行ファイルはバイナリファイル
異なるOS上でもそのまま動作する	OSごとに再コンパイルが必要
実行速度は遅い	実行速度が速い
プログラムの修正が容易	プログラムの修正は困難
エラー処理などが簡略化できる	エラー処理など、細かい処理がすべて必要
文字列の処理が容易	文字列の処理が面倒
既存のコマンドを組み合わせで一定の動作を行うのに適している	新たなアプリケーションやOS本体を記述するのに適している

シェルスクリプトとC言語の違いの実例

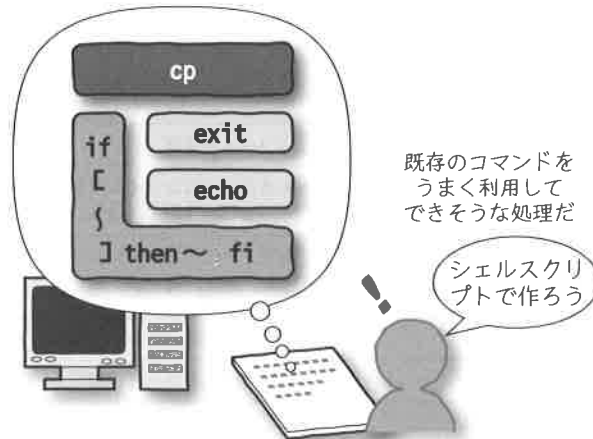
シェルスクリプトでできることは、基本的にはC言語でも記述できます。しかし、シェルスクリプトと同じ内容をC言語で記述すると、C言語のほうがかなり複雑になります。その例をリストAとリストBに示します。リストAについて、細かなことは以降で扱いますので、ここではプログラムの流れに注目してください。

このリストAとリストBは、コマンド引数で指定したファイルを、そのファイル名の末尾に「.bak」を付けたファイルにコピーするというプログラムです。たとえば、「memo.txt」というファイルなら、「memo.txt.bak」というファイルにコピーされます。

リストAのシェルスクリプトでは、一応簡単に引数の個数だけチェックした後、単にcpコマンドを呼び出すだけで完了です。cpコマンドの引数では、コマンド引数が格納された位置パラメータ"\$1"の後ろに「.bak」という文字列を付けばいいだけです。

一方、同じことをC言語で記述するとリストBのようになります。C言語では、コマンド引数をチェックした後、読み書きするファイルそれぞれをfreopen()関数を使ってオープンし、エラーチェックも行い、その後whileループでファイルの終端まで内容をコピーすることになります。また、書き込み用ファイルのファイル名として、「.bak」を付けたファイル名を作成するために、文字列のバッファを宣言し、文字列操作関数としてsprintf()を呼び出しています。ここで、このsprintf()関数の呼び出しでは、文字列の長さをチェックしていないため、あまりにも長いファイル名をコマンド引数に指定すると、sprintf()でバッファオーバーフローが発生する可能性があるという点を補足しておきます。

以上のように、ちょっとした処理でも最初からC言語で記述するとかなり面倒なことになります。シェルスクリプトにはこのような複雑さはなく、シェルスクリプトは、既存のコマンドをうまく利用し、簡単にプログラムを記述することができる言語であるといえるでしょう。



既存のコマンドを
うまく利用して
できそうな処理だ

シェルスクリ
プトで作ろう

リストA シェルスクリプトによるプログラム例

```
#!/bin/sh .....シェルスクリプトを実行するシェルを指定

if [ $# -ne 1 ]; then .....引数の個数が所定以外ならば
    echo "Usage: $0 file" 1>&2 .....エラーメッセージを出して
    exit 1 .....エラーで終了
fi

cp "$1" "$1".bak .....ファイルを「.bak」を付加したファイルにコピー
```

リストB C言語によるプログラム例

```
#include <stdio.h> .....標準入出力ライブラリ用ヘッダファイルの読み込み

int
main(int argc, char *argv[]) .....main()関数の開始
{
    int c; .....整数型変数の宣言 (後で使う)
    char filename_w[256]; .....「.bak」を付けたファイル名の作成用バッファ

    if (argc != 2) { .....引数の個数が所定以外ならば
        fprintf(stderr, "Usage: %s file\n", argv[0]); .....エラーメッセージを出して
        return 1; .....エラーで終了
    }

    if (freopen(argv[1], "r", stdin) == NULL) { .....指定ファイルが読めなければ
        perror(argv[1]); .....エラーメッセージを出して
        return 1; .....エラーで終了
    }

    sprintf(filename_w, "%s.bak", argv[1]); .....「.bak」を付けたファイル名を作成
    if (freopen(filename_w, "w", stdout) == NULL) { ... 「.bak」がオープンできなければ
        perror(filename_w); .....エラーメッセージを出して
        return 1; .....エラーで終了
    }

    while ((c = getchar()) != EOF) { .....ファイルの終端までループして
        putchar(c); .....ファイルの内容をコピーする
    }

    return 0; .....終了ステータス「0」で正常終了
}
```

Memo

●リストAは、引数チェックの必要がなければ、単に次の2行だけでも書けてしまいます。

```
#!/bin/sh
cp "$1" "$1".bak
```

シェルスクリプトの移植性

シェルスクリプトは、異なるOS上でもそのまま動作する、移植性の高いプログラム言語です。ただし、これはシェルの仕様が各OS間で統一されていることが前提です。もしも、シェルスクリプト中で使用されているコマンドや文法が、どのOS上でも同じように使えなければ、シェルスクリプトの移植性が高いとはいえなくなってしまいます。

そこで、移植性の高いシェルスクリプトを書く方法について考えてみましょう。まず基本事項ですが、シェルスクリプトの記述にはsh(/bin/sh)を用います。UNIX系OSなら、LinuxでもFreeBSDでもSolarisでも、shは標準的に存在すると考えて差し支えありません。

ここで、Linuxの多くのディストリビューションではshがbashへのシンボリックリンクになっており、shといえども実体はbashであるという点には気をつけなくてはなりません。bashはshの拡張シェルであり、shの文法はすべてbash上で使用可能ですが、逆にshでは使えない文法までがbashでは使えてしまいます。シェルスクリプト中で、bashでしか使えない文法を使用すると、ほかのOSのsh上では動作しなくなる可能性があります。たとえば**リストA**のような例です。

これらは、**リストB**のように記述を修正すればすべてのOSで動作可能になります。

このように、たとえbashを使う場合でも、移植性を第一に考えるならば従来のshの文法の範囲のみを使うべきでしょう。

移植性の高いシェル文法の判断

それでは、どの文法までなら使ってよいのかを判断する方法を考えてみましょう。現実的な方法としては、bashのほかに代表的なOSとして、FreeBSDのshとSolarisのshを用い、これらのシェルでも同じように動作するかどうかをチェックし、これを判断基準とするのが実践的でしょう。

本書では実際に各項目すべてについて、Linux(bash)、FreeBSD(sh)、Solaris(sh)にて動作確認を行い、その結果をアイコンで表示しています。3種類のシェルすべてのアイコンに○が表示されている項目は、移植性の高いシェル文法として安心して使ってよいでしょう。

リストA bash、FreeBSDのshで動作可能、Solarisのshで動作不可能な例

```
export LANG=C .....環境変数LANGにC（国際化前の言語）を設定
dir=$(pwd) .....カレントディレクトリ名をシェル変数dirに代入
```

リストB bash、FreeBSDのsh、Solarisのshのすべてで動作可能な例

```
LANG=C; export LANG .....環境変数LANGにCを設定
dir='pwd' .....カレントディレクトリ名をシェル変数dirに代入
```

Memo

- Solarisのshは拡張がほとんどなく、より従来のshに近い動作をするため、リファレンス用シェルとして便利です。
- Linuxのashは、共通の由来を持つFreeBSDのshに近い動作をするため、手元にFreeBSDがない場合でも、Linuxのashを使って簡単な移植性チェックができます。

>第1章

シェルスクリプト入門

1.1	Hello World	14
1.2	#!/bin/shの意味.....	16
1.3	実行方法について	18

シェルスクリプト作成の流れ

- Linux (bash)
- FreeBSD (sh)
- Solaris (sh)

Hello Worldを書いてみよう

例 `#!/bin/sh` 1行目の行頭から、`#!`に続いてシェルの絶対パスを書く
 ◀ わかりやすいように1行空ける (空けなくてもよい)
`echo 'Hello World'` echoコマンドで任意の文字列を表示する

解説 記述形式の基本

まずは手始めに、「Hello World」というメッセージを表示するだけの簡単なシェルスクリプトを作成してみましょう。その「Hello World」シェルスクリプト(ファイル名「hello」)が冒頭の例です。

1行目には`#!/bin/sh` というように、1行目の行頭から`#!`と書き、これに続いてシェルの絶対パスである`/bin/sh`を記述します。シェルスクリプトは基本的にこの「`#!/bin/sh`」の行で始まります。

2行目は、わかりやすいように1行空けていますが、これは空けなくてもかまいません。シェルスクリプトではこのような空行は単に無視されます。

3行目には`echo` コマンドを記述しています。この`echo` コマンドがこのシェルスクリプトの本体になります。`echo` は、引数で指定された文字列を単に標準出力(通常は画面)に出力するというコマンドです。この例では引数で「Hello World」という文字列を指定しているため、そのまま「Hello World」と画面に表示されるはずです。

なお、「Hello World」の文字列全体をシングルクォート(' ')で囲んでいるのは、文字列中にもしも特殊な文字が含まれていた場合に、それがシェルによって解釈されるのを防ぐためです。

シェルスクリプトの作成と実行

実際にシェルスクリプトを作成し、実行している様子を図Aに示します。このように、まずはテキストエディタで「hello」というファイルを新規作成し、リストAの内容(冒頭の例と同じ)を入力してください。図A①ではviエディタを使用していますが、KDE付属のKEdit、GNOME付属のgeditなど、テキストエディタであればなんでも使用できます。ただし、改行コードはLFのみになるようにしてください^{注1}。

ファイルが作成できたら、「`chmod +x` (ファイル名)」というコマンドでファイルに**実行属性を付加**します(図A②)。これで「hello」というファイルが、実行可能なシェルスクリプトになりました。

注1 LinuxなどのUNIX系OSでは、テキストファイルの改行コードは標準でLFのみになるため、とくに気にする必要はありませんが、Windowsでテキストファイルを作成する場合、標準の改行コードがCR+LFになってしまうため、注意してください。

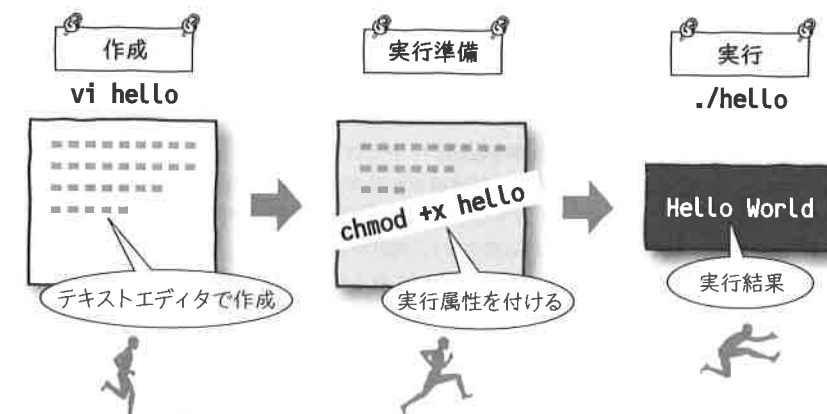
この「hello」を実行するには、図A③のように、頭に`./`を付けて`./hello`とコマンド入力します。すると、「Hello World」というメッセージが表示され、たしかにシェルスクリプトが実行されたことがわかります。なお、ここで頭に`./`を付ける必要があるのは、安全のため、カレントディレクトリに実行パスを通してないからです。この「hello」というシェルスクリプトを、たとえば`/usr/local/bin`などのような実行パスの通ったディレクトリにインストールすれば、頭に`./`を付けずに単に「hello」というコマンド名で実行できるようになります^{注2}。

図A シェルスクリプトhelloの作成と実行

<code>\$ vi hello</code>	①テキストエディタでリストAの内容の新規ファイルを作成
<code>\$ chmod +x hello</code>	②作成したファイルに実行属性を付ける
<code>\$./hello</code>	③このシェルスクリプトを実行する
Hello World	たしかにHello World と表示される

リストA hello(シェルスクリプトによるHello World)

```
#!/bin/sh
echo 'Hello World'
```



参照

echo(p.137)

シングルクォート'(p.207)

注2 `/bin`、`/usr/bin`、`/usr/local/bin`などのディレクトリの下にインストールされているコマンドは、そのコマンド名を入力しただけで実行できます。これは、`/bin`などがあらかじめ実行パスに登録されているからです。現在設定されている実行パスは、`echo $PATH` というコマンドを実行すれば確認できます。

#!/bin/shの意味



シェルスクリプトの1行目

例 `#!/bin/sh` 1行目の行頭から、#!/に続いてシェルの絶対パスを書く
`echo 'Hello World'`

..... ファイル名は「hello」にしておく

解説

シェルスクリプトの1行目には通常、#!/bin/shと書きます。このように書いておくと、シェルスクリプトの実行時に、システム内部で/bin/sh [シェルスクリプト名]というコマンドが実行されます。つまり、冒頭の「hello」というシェルスクリプトの場合、./helloとコマンド入力した時に、システム上では/bin/sh ./helloというコマンドが実行されているのです。

/bin/shは、「hello」というファイルの中身を1行ずつ実行しますが、この時、1行目の#!/bin/shの行自体は、行頭に#があるため、シェルにとってはコメントとみなされ、単に無視されます。

#!/bin/sh以外のスクリプト(Perlやawk)

1行目の#!/bin/shの部分に、/bin/sh以外のコマンドを書いてもかまいません。たとえば#!/usr/bin/perlと書けばPerlスクリプトに、#!/usr/bin/awk -fと書けばawkスクリプトになります(awkの場合は-fオプションが必要です)。Perlやawkを使用したHello WorldのスクリプトをそれぞれリストA、リストBに示します。これらのスクリプトも、chmod +xで実行属性を付けて実行すれば、もちろん「Hello World」と表示されます。この時、システム上ではそれぞれ/usr/bin/perl ./perl_hello」「usr/bin/awk -f ./awk_hello」というコマンドが実行されています。

リストA perl_hello(PerlによるHello World)

`#!/usr/bin/perl` Perlスクリプトであることを明示
`print "Hello World\n";` Perl文法によるprint行

リストB awk_hello(awkによるHello World)

`#!/usr/bin/awk -f` awkスクリプトであることを明示 (-fオプションが必要)
`BEGIN {`
`print "Hello World"` awk文法によるprint行
`}`

tailコマンドを使った変わったスクリプト

さらに、1行目の#!/のところにPerlでもawkでもない、普通のコマンドを書くこともできます。一般に、#!/の右にどのようなコマンドを書いても、そのスクリプトの実行時に、単に#!/のところに書いたコマンドが、スクリプトのファイル名を引数に付けて実行されるにすぎないのです。

リストCはtailコマンドを利用した変わったHello Worldです。この「tail_hello」を実行すると、システム上では「/usr/bin/tail -1 ./tail_hello」というコマンドが実行され、tailコマンドの-1オプションにより、tail_helloファイルの最後の1行が表示されることになります。tail_helloの最後の1行には、直接「Hello World」と書いてあるため、この文字列が直接表示されるというしくみです。

リストC tail_hello(tailによるHello World)

`#!/usr/bin/tail -1` tail -1を実行するスクリプトであることを明示
`.....` 2行目は空行
`Hello World` 最後の1行に直接メッセージを記述

Memo

① 1行目の#!/のところに書くコマンドに対する引数は、1個のみ記述するようにしてください(awkの-fやtailの-1など)。2個以上の引数を記述した場合の動作は、次に示すとおりOSによって異なります。

- Linuxの場合：すべての引数がスペースを含めてつながり、全体が1個の引数とみなされる
- FreeBSDの場合：Linuxと同様に、すべての引数がスペースを含めてつながり、全体が1個の引数とみなされる(ただしFreeBSD 5.x以前では2個以上の引数を使用することが可能)
- Solarisの場合：記述された引数のうち、1個目のみが認識される

② 1行目の#!/のところに別のシェルスクリプト等を記述しても正しく実行されません。ここには、実行バイナリファイルを記述する必要があります。

参照

コメントの書き方(p.23)

いろいろな実行方法

コマンドとして実行する以外には

- Linux (bash)
- FreeBSD (sh)
- Solaris (sh)

解説

#!/bin/shの行やchmod +xの必要性

シェルスクリプトには通常、#!/bin/shの行と、chmod +xコマンドによる実行属性の付加が必要です。しかし、これらが本当に必要なのは、シェルスクリプトを通常のコマンドとして実行する場合のみです。

シェルスクリプトのいろいろな実行方法

シェルスクリプトには、コマンドとして実行する以外に、自分でshの引数に指定して実行する方法など、いくつかの別の実行方法があります。これらを表Aにまとめます。それぞれ微妙に実行方法やその動作が異なりますので、参考にしてください。ここでは実行するシェルスクリプトのファイル名をshfileとしています。

表A シェルスクリプト(ファイル名:shfile)のいろいろな実行方法

	実行方法	#!/bin/shの行	chmod +x	実行パス	実行シェル	引数の付け方
コマンドとして実行	\$ shfile ^{※1}	通常は必要 ^{※2}	必要	参照する	新しいシェル	\$ shfile (引数1) (引数2) ...
シェルの引数として実行	\$ sh shfile	不要	不要	参照しない	新しいシェル	\$ sh shfile (引数1) (引数2) ...
標準入力を実行	\$ sh < shfile	不要	不要	参照しない	新しいシェル	\$ sh -s < shfile (引数1) (引数2) ... ^{※3}
. コマンドで実行	.\$ shfile	不要	不要	参照する	現在のシェル	(引数は付けられない) ^{※4}

- ※1 実行パスの通っていないカレントディレクトリ上のシェルスクリプトを実行する場合は\$./shfileとして実行する
- ※2 #!/bin/shの行がないと、カーネルレベルでのexecにいったん失敗した後、シェルの判断によって/bin/shのシェルスクリプトであるとみなされて/bin/sh shfileが実行される
- ※3 shに-sオプションを付けて標準入力を実行することを明示する。-sオプションがないと引数1がシェルスクリプトのファイル名であると誤って判断されてしまう
- ※4 bashの場合は、コマンドに対して引数を指定することもできる

参照

. コマンド (p.90)

> 第2章

シェルスクリプトの基本事項

- 2.1 シェルスクリプトはフリーフォーマット20
- 2.2 コメントの書き方23
- 2.3 コマンドの終了ステータス25

シェルスクリプトは フリーフォーマット



改行位置やインデントなどはかなり自由

解説

記述の基本

シェルスクリプトは、それ自体がソースファイルであると同時に**実行ファイル**です。シェルは、シェルスクリプトの記述内容を逐次読み込みながらコマンド実行を進めます。しかし、実行ファイルであっても、その**改行位置やインデントの仕方**はかなり自由であり、さらにコメントを書き込むこともできるため、人間が読んでもわかりやすいシェルスクリプトを記述することができます。また、シェルスクリプト中の各コマンドはすべて**終了ステータス**を持っており、シェルスクリプトでは基本的にこの終了ステータスの真偽によって各種条件判断を行います。

シェルスクリプトはフリーフォーマット

前章のとおり、シェルスクリプトは、基本的にはコマンドラインに入力するコマンドの文字列をファイルに記述したものです。したがって、その記述は1行単位の固定されたフォーマットであると思われるかもしれません。しかし、シェルスクリプトでは記述が見やすいように、任意に**スペース、タブ、改行**を入れることができます^{注1}。とくに、コマンドの行頭にスペースやタブを入れる**インデント**を行うことにより、if文、for文などの構文の構造が把握しやすくなります。ただし、改行については、改行コードにリストを終端する意味があるため、任意の位置でまったく自由に改行してよいわけではなく、多少の制限があります^{注2}。これは、改行もスペースと同様に扱うC言語とは異なる点といえるでしょう。

行頭にスペースまたはタブを入れてもよい

コマンドの入力時には、コマンド名の前に、好きなだけスペースやタブを入れることができます。**図A**は、シェルのプロンプト上に直接lsコマンドを入力している例ですが、このようにスペースやタブがあっても、シェルにとってはすべて同じlsコマンドとして解釈されます。

図A lsコマンドの実行情例1

\$ ls	普通にlsコマンドを入力
\$ ls	lsコマンドの前にスペース2つを追加
\$ ls	lsコマンドの前にタブ1つを追加

注1 bashのコマンドラインに直接コマンド入力する場合、タブは補完用のキーに割り当てられているため、普通のタブを入力するには`(Ctrl)+[V]`、`(Tab)`とタイプする必要があります。

注2 「リストを終端する」とは、「そこまでの入力コマンドとして実行されること」と、とりあえず考えておいて差し支えありません。詳しくはリストの項(p.35)を参照してください。

行の継続で、コマンドの途中で改行してもよい

さらにシェル文法上では、「**改行**」のようにバックスラッシュ(\)の直後で改行すると、そこで1行につながり、バックスラッシュも改行もなかったものとして解釈されます。これを利用して、**図B**のように、lsコマンドのlとsの間で改行を行えます。奇妙ですが、これもlsコマンドとして解釈されるのです。なお、改行の直後の2行目では、まだコマンドが完結していないという意味で、シェルのプロンプトがプライマリ「\$」からセカンダリの「>」に変わります。

改行してもつながるパイプ

lsコマンドの直後に改行すると、そこでlsコマンドが実行されてしまいますが、**図C**のようにlsの後に**パイプ**の|記号までを入力してから改行すると、シェルはまだパイプラインの途中であると解釈するため、セカンダリプロンプトを出して入力待ちになります。ここに、lessなどのコマンドを入力すると、無事パイプにつながり、結局1行で「ls | less」と入力したのと同じことになります。

この例での|記号の直後のように、リストの終端とはみなされない位置で改行した場合、その改行はスペースなどと同じ単なる区切り文字とみなされるのです。

なお、パイプのほかにも、**&&リスト**や**||リスト**などでも、&&や||の直後で任意に改行することができます。

いろいろなif文の書き方

if文、while文などの各種構文は、その改行位置の違いによっていろいろな書き方ができます。ここでは、if文を例にとってそのいろいろな書き方を紹介しておきます。

リストA~リストDは、いろいろな書き方をしたif文です。もちろん、これらはすべて同じ意味になります。

リストAは標準的な書き方です。ifの右のtestコマンド([])の後ろで改行した後、thenの所でも改行し、if文の中のechoコマンドではインデントしています。**リストB**はやや短縮した書き方です。ifとthenを1行に書いていますが、thenの直前のtestコマンドのリストを終端する必要があるため、ここに;を入れています。**リストC**のような書き方はあまり行いませんが、thenの後にはもともと改行が必要ないことを利用し、thenの後にechoコマンドを1行で続けています。このechoの前には;を入れてはいけません。**リストD**はif文全体を1行に書いた例です。thenの前やfiの前では、その直前のリストを終端するために;が必要になります。短いif文の場合、1行に書いたほうが簡潔で良い場合があります。

図B lsコマンドの実行情例2

\$ ls \	lsコマンドのlの1文字だけで、「改行」で改行
> s	セカンダリプロンプトに続いてlsのsを入力

図C 改行してもつながるパイプ

\$ ls	パイプの の直後に改行
> less	セカンダリプロンプトに続いてlessを入力すればパイプがつながる

コメントの書き方



コメントを記入するには#記号を使う

書式 `[文字列] ... # コメント`

例 `echo 'Hello' #メッセージを表示`#から行末まではコメントとして無視される

基本事項

#が単語の1文字目として使用されている場合(スペースまたはタブの区切り文字の直後、または行頭に#がある場合)、#に続く `コメント` 文字列は#から行末までコメントとして無視されます。

解説

シェルスクリプトにおける **コメント** の記号は#です。基本的には#から行末までがシェルによって無視され、ここにシェル文法とは無関係に任意の文字を記述できます。

ただし、#は、**行頭**か、**スペース**または**タブ**の区切り文字の直後に記述する必要があります。文字列の途中に#がある場合はコメントとはみなされません。また、シングルクォート(')、ダブルクォート(")、バックスラッシュ(\)で#がクォートされた場合も、コメントにはなりません。

なお、シェルスクリプトの1行目に記述する#!/bin/shは、シェルにとっては単なるコメントとして解釈されるということは、先に説明したとおりです。

コメントとはみなされない#について

図A①のように「Hello#World」という文字列を使用した場合、#はスペースなどの直後にはないため、コメントとはみなされず、#を含んだ文字列全体がechoコマンドの引数であると解釈されて実行されます。

また、図A②の\$#についても、同じく#がスペースなどの直後にはないため、コメントとはみなされず、特殊パラメータ\$#として、現在の位置パラメータ(引数)の個数(図Aでは0個)に展開されます。

図A コメントとはみなされない#の実行例

\$ echo Hello#World	①文字列の途中に#がある場合
echo Hello#World	#を含めたすべての文字列が表示される
\$ echo 引数は \$# 個です	②文字列の途中であり、かつ特殊パラメータの場合
引数は0個です	\$#が0に展開されて表示される

リストA if文の記述例1

```
if [ -f file ] .....ifの右のtestコマンド([ ])の後ろで改行
then .....thenのところで改行
echo 'fileが存在します' .....if文の中のechoコマンドはインデントして記述
fi .....if文の終了
```

リストB if文の記述例2

```
if [ -f file ]; then .....;を使って、ifとthenの1行で記述
echo 'fileが存在します' .....if文の中のechoコマンドはインデントして記述
fi .....if文の終了
```

リストC if文の記述例3

```
if [ -f file ] .....ifの右のtestコマンド([ ])の後ろで改行
then echo 'fileが存在します' .....thenの後ろにechoコマンドを続ける(;は不要)
fi .....if文の終了
```

リストD if文の記述例4

```
if [ -f file ]; then echo 'fileが存在します'; fi .....if文全体を1行で記述
```

改行の仕方のまとめ

シェルスクリプトでの改行の仕方について、簡単にまとめると次のようになります。

●リストを終端する必要がある場合は改行するか、代わりに;を使う

if文のthenの前やwhile文のdoの前など、リストを終端する必要がある場合は基本的にはそこで改行します。1行に書きたい場合は改行の代わりに;を入れる必要があります。

●リストの終端でない位置には任意に改行を入れてよい

if文のthenの直後やパイプの|の直後など、改行してもリストの終端とはみなされない位置では任意に改行を入れることができます。ただし、逆に1行に書く場合には;は不要で、;を入れるとエラーになります。

●本来改行を入れることができない位置でも、「\」で行の継続を行うことができる

本来改行を入れることができない位置、たとえばコマンド引数などの途中でも、バックスラッシュ(\)の直後に改行することによって、2行以上に分けて記述することができます。

参照

リスト(p.35) バックスラッシュ\ (p.211) パイプライン(p.32) if文(p.43)

コメントをつけられない行について

リストAでは、echo コマンドとその引数との間に\改行を入れ、echo コマンドを2行に分けて記述しています。この場合、行の継続を行うために、1行目の\の直後に改行を入れなければならないため、\の右側に#などを追加してコメントを書くことはできません。コメントは、2行目の引数の後に記述するようにします。

リストA コメントをつけられない行の例

```
echo \ .....この行にはコメントがつけられない
'Hello World' #コメント .....この行にはコメントがつけられる
```

注意事項

#の直前にはスペースを

行頭以外にコメントを書く場合、#の直前にスペースを入れることを忘れないください。次の誤った例のように、コマンドの直後に#を続けて書くと、コメントとはみなされなくなってしまいます。

○正しい例

```
echo 'Hello World' #コメント.....#の前にスペースがあるのでコメントとみなされる
```

×誤った例

```
echo 'Hello World'#コメント.....#の前にスペースがないのでコメントとはみなされない
```

参照

シングルクォート' (p.207) ダブルクォート" (p.209) バックスラッシュ\ (p.211)
特殊パラメータ\$# (p.171)

コマンドの終了ステータス



条件判断を行う際に利用される

表

終了ステータス	真偽値
0	真
0以外	偽

基本事項

シェルスクリプトでは、各コマンドの**終了ステータス**が**0**ならば**真**、**0以外**ならば**偽**の意味であると判定されます。

解説

シェルスクリプトでは、各コマンドの実行の際に、その**終了ステータス**を参照することによって、if文、while文、&&リスト、||リストなどの**条件判断**を行います。

コマンドが**単純コマンド**かつ**外部コマンド**の場合^{注3}、その外部コマンドのプログラムがexitシステムコールでOSに返す値が終了ステータスになります。これは、プログラムがC言語で書かれている場合は、exit()関数の引数の値またはmain()関数のreturnの値になります。

コマンドが組み込みコマンドや、構文などの**複合コマンド**の場合は、終了ステータスはそれぞれの文法で決められています。

いずれも、終了ステータスが**0**の場合は**真**、**0以外**ならば**偽**となります。これはC言語のif文などでの真偽判定とは逆になっているので注意してください。「0」を真としているのは、通常のコマンドは正常終了時に「0」を返し、エラー時に「0」以外のエラーコードを返すので、「0」を真としたほうが都合がよいからです。

シェルスクリプトでよく使用するtestコマンドは、画面には何も表示せず、もっぱら終了ステータスを返すという重要なコマンドです。また、単に真または偽の終了ステータスを返すだけが目的の、true、falseというコマンドも存在します。

なお、終了ステータスは、if文などで直接使用するほかに、コマンドの実行直後に特殊パラメータ\$?を参照することによっても得られます。

また、シェルスクリプト自体の終了ステータスは、exitコマンドによって返せます。

終了ステータスの利用

終了ステータスを利用した実行例を図Aに示します。ここでは、シェルスクリプトではなく、シェルのプロンプト上に直接コマンドを入力しています。図A①②のように、trueやfalseコマンドの直後に特殊パラメータ\$?を参照すると、それぞれ「0」や「1」の値が表示されることがわかります。

注3 外部コマンドについては、p.260(図A)、12章を参照してください。

また、図A③のようにif文にtrueコマンドを使用すると、無条件でthenからfiまでのリストが実行されることがわかります。なお、ここでif文の2行目以降の入力時に、シェルのプロンプトがプライマリの「\$」からセカンダリの「>」に変わり、if文の構文の途中であることが示されている点に注意してください。

図A 終了ステータスの利用例

\$ true	①trueコマンドを実行
\$ echo \$?	直後に特殊パラメータ\$?を参照すると
0	真を意味する0が表示される
\$ false	②falseコマンドを実行
\$ echo \$?	直後に特殊パラメータ\$?を参照すると
1	偽を意味する1が表示される
\$ if true	③if文の条件判断にtrueコマンドを使用
> then	真ならばthen以下が実行される
> echo '真です'	echoコマンドを記述
> fi	if文の終了
真です	たしかに、記述したechoコマンドが実行される

Memo

- コマンドの終了ステータスは、シェルスクリプト以外にも、make コマンドでコンパイルエラーが発生した時に make を中断するなどの目的でも使用されています。

参照

if文(p.43)	while文(p.63)	&&リスト(p.37)	リスト(p.39)
単純コマンド(p.29)	複合コマンド(p.30)	test(p.147)	true(p.150)
false(p.140)	特殊パラメータ\$(p.173)	exit(p.103)	

> 第3章

シェル文法の循環構造

3.1	コマンド・パイプライン・リストの循環	28
3.2	コマンド／パイプライン／リスト	29
3.3	&&リスト／ リスト	37

コマンド→パイプライン→リストの循環

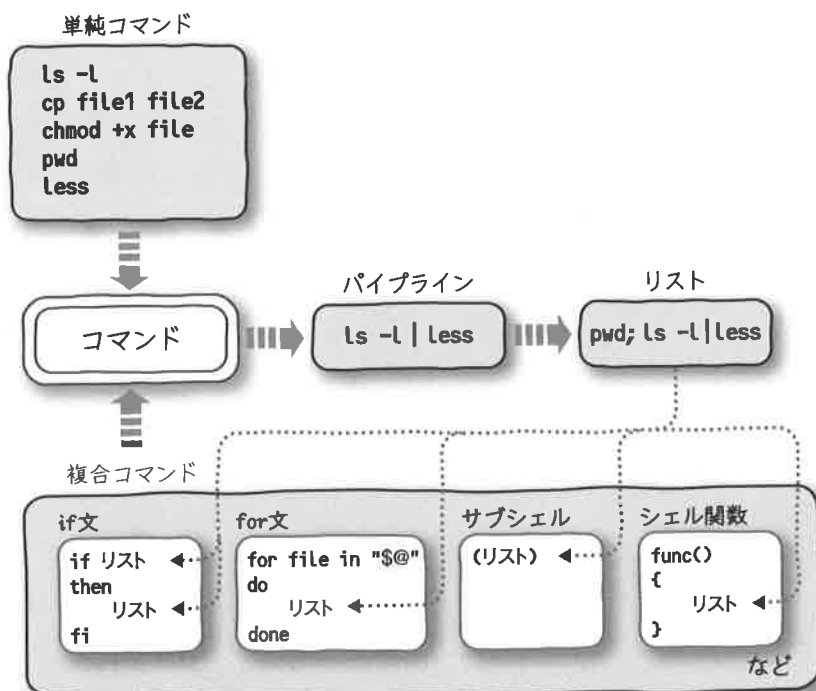
シェル文法の解説では、単純コマンド、複合コマンド、コマンド、パイプライン、リストといった用語が登場します。これらは、図Aのような循環構造を構成しています。

たとえば、lsやcpなどの、普通の意味で「コマンド」と呼んでいるものは、シェル文法上では**単純コマンド**と呼ばれます。一方、if文、for文などの構文や、()で囲んだサブシェル、シェル関数などのことを**複合コマンド**と呼びます。そして、単純コマンドと複合コマンドを合わせて**コマンド**と呼びます。つまり、シェル文法上ではif文などの構文もそれ全体が1つのコマンドなのです。

さらに、1つ以上のコマンドがパイプ(|)でつながったものが**パイプライン**になり、1つ以上のパイプラインがセミコロン(;)または改行などでつながったものが**リスト**になります。

このリストは、改めて別の複合コマンドの内部の要素になります。たとえば、if文のifやthenの直後に記述するリストとして使うことができます。この結果、**コマンド→パイプライン→リスト→複合コマンド→コマンド**という循環構造ができあがることになるのです。詳しくは本章の単純コマンド、複合コマンド、コマンド、パイプライン、リストの各項を参照してください。

図A シェル文法の循環構造



単純コマンド



一般のUNIXコマンドは単純コマンドとして実行できる

書式 **コマンド名** [**引数** ...]

例
 ls -l カレントディレクトリのファイルのリストの表示
 echo 'Hello World' メッセージの表示
 pwd カレントディレクトリ名の表示

基本事項

[コマンド名]と0個以上の[引数]を、スペースなどの区切り文字で区切って並べたものが**単純コマンド**です。単純コマンドは単に実行されます。

終了ステータス

実行されたコマンドの終了ステータスそのものが、単純コマンドの終了ステータスです。

解説

シェル文法上の単純コマンドは、普通の意味での「コマンド」です。つまり、ls、cp、chmodなどのいわゆるUNIXコマンドはすべて単純コマンドです(p.31のコラム参照)。引数は、付けても付けなくてもかまいません。シェルスクリプト中に単純コマンドを記述すると、その名前のコマンドが実行されます。単純コマンドは、ls、cpなどのように**外部コマンド**として実装されている場合と、cd、echoなどのようにシェルの**組み込みコマンド**(内部コマンド)として実装されている場合があります。

リダイレクトと、環境変数の一時変更

単純コマンドの実行時に、標準入出力などをファイルにリダイレクトしたり、環境変数の一時変更を行うこともできます^{注1}。リストAでは、単純コマンドであるdateコマンドに対して、環境変数LANGの一時変更と、その標準出力のファイルへのリダイレクトを行っています。dateは日時を表示するコマンドです。ここではLANG=Cと記述されていることから、日時の英語表記が「timestamp」というファイルに書き込まれることになります。

リストA リダイレクトと、環境変数の一時変更

LANG=C date > timestamp 環境変数LANGを一時的にCに変更し、dateの標準出力をファイルにリダイレクトする

注1 リダイレクトについては11章、環境変数の一時変更についてはp.180を参照してください。

複合コマンド

構文／サブシェル／シェル関数などは
複合コマンドとして解釈される



書式 **if文** **case文** **for文** **while文** **サブシェル** **グループコマンド** **シェル関数**

例 while ;; do 無限ループのwhile文の開始
echo 'Hello World' メッセージの出力
done while文の終了 (ここまで全体が1つの複合コマンド)

基本事項

複合コマンドとは、**if文** **case文** **for文** **while文** **サブシェル** **グループコマンド** **シェル関数**のことです。
なお、複合コマンドについては4章で詳しく扱います。

終了ステータス

複合コマンドの終了ステータスは、各複合コマンドの仕様によって決められています。

解説

シェル文法上では、サブシェルやシェル関数などはもちろん、if文やfor文などの構文も、それ全体が1つの**複合コマンド**であると解釈されます。複合コマンドは、単純コマンドと合わせて**コマンド**と呼ばれます。

複合コマンドのリダイレクト

複合コマンドは全体が1つのコマンドなので、この標準出力をファイルにリダイレクトすることができます。たとえば**リストA**のように記述すれば、for文全体の標準出力が「newfile」というファイルにリダイレクトされ、このファイル中には「Hello World」の文字が5行分、書き込まれます。リダイレクトについては12章を参照してください。

リストA 複合コマンドのリダイレクト

```
for n in 1 2 3 4 5
do .....for文による5回ループ
echo 'Hello World' .....メッセージの出力
done > newfile .....以上をnewfileというファイルに書き込む
```

bashの場合の複合コマンド

bashでは、さらに算術式のfor文(p.61)、select文(p.69)、算術式の評価(())(p.80)、条件式の評価[[]](p.83)についても文法上、複合コマンドと解釈されます。



Warning
Linux (bash) x FreeBSD (sh) x Solaris (sh)
この方法には制限があります。

コマンド

単純コマンドと複合コマンドとを
合わせてコマンドと呼ぶ



書式 **単純コマンド** **複合コマンド**

例 ls -llsコマンド自体が1つのコマンド
while ;; do無限ループのwhile文の開始
echo 'Hello World'メッセージの出力
donewhile文の終了 (while文全体が1つのコマンド)

基本事項

コマンドとは、**単純コマンド**または**複合コマンド**のことです。

終了ステータス

コマンドの終了ステータスは、単純コマンドまたは複合コマンドの仕様によって決められています。

解説

シェル文法上では、lsやechoなどの単純コマンドと、if文やサブシェルなどの複合コマンドをすべて合わせて**コマンド**と呼びます。コマンドは、パイプ(|)でつないで**パイプライン**を構成できます。

Column

「いわゆる」UNIXコマンド(単純コマンド)の調べ方

ユーザが使用できるUNIXコマンドは、/bin、/usr/bin、/usr/local/binといったディレクトリにインストールされているのが普通です。これらのディレクトリにあるファイルは、コマンドとして実行することができます。これらのディレクトリにパスが通っていれば、たとえばlsコマンドなら、type lsというコマンドを実行してlsコマンドの存在を確認できます。

なお、コマンドのオプションなどの詳しい使い方については、man lsのように実行すればオンラインマニュアルが読めますので、必要に応じて参考にしてください。

パイプライン

コマンドの標準出力を別のコマンドの標準入力に接続する



書式 コマンド | コマンド...

例 ls -l | less lsコマンドの出力をパイプでlessに接続して表示する

基本事項

パイプラインとは、1つ以上のコマンドを、パイプ(|)で区切って並べたもののことです。左側のコマンドの標準出力は、右側のコマンドの標準入力に、パイプで接続されます。

終了ステータス

パイプライン中の最も右側(最後)のコマンドの終了ステータスが、パイプラインの終了ステータスになります。

解説

複数のコマンドを、パイプの記号である|で区切って並べるにより、パイプラインを構成することができます。ここで、コマンドとは単純コマンドまたは複合コマンドを意味するため、lsやechoなどの単純コマンドだけでなく、while文やサブシェルなども、パイプで接続してパイプラインにすることができます。なお、コマンドが1つだけ(パイプは0個)であっても、シェル文法上はパイプラインと呼びます。パイプラインを改行や;で区切って並べるとリストになります。

パイプラインでは、左側のコマンドの標準出力がパイプを通過して右側のコマンドの標準入力に入力されるため、各種フィルタコマンドを部品のように組み合わせて各種処理を行えます。

パイプラインの終了ステータスの利用

パイプラインの終了ステータスは、パイプライン中の最後のコマンドの終了ステータスです。したがって、図Aのようにwhoコマンドとgrepコマンドをパイプでつなぐと、grepコマンドの終了ステータスがパイプラインの終了ステータスになります。ここでは、「guest」というユーザがログインしていれば、このパイプラインが「真」になります。図Aのように直後に特殊パラメータ\$?の値を表示して、終了ステータスを知ることができます。

図A パイプラインの終了ステータスの利用

```
$ who | grep -q '^guest\>'    guestがログインしているかチェック (画面は非表示)
$ echo $?                    終了ステータスを表示
0                             0 (真) が表示されたのでguestがログインしている
```

パイプラインの否定演算

bashまたはFreeBSDのshでは、図Bのように、パイプラインの先頭に!を記述して、パイプラインの終了ステータスを反転することができます。これは、if文の条件判断を逆にする目的でも使えます。

なお、履歴の!と区別するため、!とコマンドとの間にはスペースが必要です。

図B パイプラインの否定演算

```
$ who | grep -q '^guest\>'    guestがログインしているかチェック (画面は非表示)
$ echo $?                    終了ステータスを表示
0                             0 (真) が表示されたのでguestがログインしている
$ ! who | grep -q '^guest\>'  同じパイプラインの頭に否定演算の!を付ける
$ echo $?                    終了ステータスを表示
1                             条件が逆になり、1 (偽) が表示される
```

パイプラインのコマンドの終了ステータスを個別に得る方法

bashでは、パイプラインを構成する各コマンドの終了ステータスを個別に取得することができます。

パイプラインを実行すると、各コマンドの終了ステータスはPIPESTATUSという配列型のシェル変数に代入されます。パイプラインの左端のコマンドの終了ステータスはPIPESTATUS[0]に、以下右に向かって順にPIPESTATUS[1]、PIPESTATUS[2]...と代入されます。

図Cは、PIPESTATUSに代入される様子を実際に確認している例です。パイプライン中の(exit 3)は、サブシェルを利用して終了ステータス3を返すというコマンドです。(exit 4)と(exit 5)も同様です。このパイプラインを実行した直後、配列の内容を一括して参照する\${PIPESTATUS[@]}という記法を使って、その値を表示しています。

なお、PIPESTATUSの値はパイプラインを実行するたびに書き換えられます。echoコマンド自体も(コマンド1個だけの)パイプラインであるため、たとえばecho \${PIPESTATUS[0]}を実行してしまうと、次にecho \${PIPESTATUS[1]}を実行しても値は失われてしまいます。このため、PIPESTATUSの値は\${PIPESTATUS[@]}で一括して参照する必要があります。PIPESTATUSの値を温存して個別に参照したい場合は、array=(\${PIPESTATUS[@]})のように、いったん別の配列変数に一括代入した後、\${array[0]}、\${array[1]}...などで参照するようにします。

図C パイプラインのコマンドの終了ステータスを個別に表示

```
$ (exit 3) | (exit 4) | (exit 5)  終了ステータス3、4、5を返すパイプラインを実行
$ echo ${PIPESTATUS[@]}          配列型のシェル変数PIPESTATUSの内容を一括表示
3 4 5                            確かに各コマンドの終了ステータスが表示される
```


Memo

- パイプライン中の各コマンドは、OS上で並行処理で同時に実行され、また、パイプを通る標準入出力のデータは、テンポラリファイルを使用せずに受け渡されるため、パイプラインは非常に効率のよい実行方法です。

3

2

コマンド／パイプライン／リスト

参照

特殊パラメータ \$(p.173) if文(p.43)

リスト

- Linux (bash)
- FreeBSD (sh)
- Solaris (sh)

パイプラインを改行などで区切って並べたもの

書式 `パイプライン` `改行` ; `&` `&&` || `パイプライン` ... `改行` ; `&`

例 `cd /some/dir; ls -l cdとlsという2つのパイプラインが;で区切って並べられ、1つのリストになっている`

基本事項

リストとは、1つ以上の`パイプライン`を、`改行`、`;`、`&`、`&&`または`||`で区切って並べたもののことです。リストの最後には、`改行`、`;`または`&`をつけることができ、これらがついたリストのことを「終端されたリスト」と呼びます。

`パイプライン`が、`改行`または`;`で区切られているかまたは終端されている場合、各`パイプライン`は左から右に順番にフォアグラウンドで実行されます。`パイプライン`が、`&`で区切られているかまたは終端されている場合、`&`の左の`パイプライン`はバックグラウンドで実行されます。`パイプライン`が`&&`または`||`で区切られている場合の動作は、それぞれ`&&`リスト、`||`リストの項で解説します。

なお、`&&`と`||`は、`改行`、`;`、`&`よりも高い優先順位で評価されます。

終了ステータス

リストの中で最後に実行したパイプラインの終了ステータスが、リストの終了ステータスになります。ただし、最後のパイプラインがバックグラウンドで実行された場合は終了ステータスは「0」になります。

解説

複数のパイプラインを改行などで区切って並べれば、リストになります。パイプラインには単純コマンドも含むため、結局「`ls -l`」`cp file1 file2`のような、通常のコマンドライン上のコマンド入力も、それ全体が1つのリストということになります。

リストは、`if`文/`while`文などの構文や、サブシェル、コマンド置換その他、シェル文法上でリストが使用できるとされている場所に用いることができます。

終端されたリストと終端されていないリスト

`if`文/`while`文などの構文のリストなどでは、リストが(フォアグラウンドの場合)改行または`;`で終端されている必要がありますが、サブシェルやコマンド置換ではリストが終端されていなくてもかまいません。具体的にはリストAのように、`if`文の`then`の直前には改行または`;`が必要ですが、サブシェルでは閉じカッコ()`()`の前に改行や`;`がなくてもかまいません。

3

2

コマンド／パイプライン／リスト

&&リスト



簡単な条件判断を行える

書式 **パイプライン1 && パイプライン2**

例 `test -f file1 && cp file1 file2`file1が存在すればfile2にコピーする

基本事項

&&リストでは、まず「パイプライン1」が実行され、その結果が真(終了ステータスが「0」)である場合のみ「パイプライン2」が実行されます。

終了ステータス

パイプライン1が真でパイプライン2が実行された場合はパイプライン2の終了ステータス、パイプライン1が偽の場合はパイプライン1の終了ステータスが、&&リスト全体の終了ステータスになります。

解説

&&リストの本来の意味は、&&の左右のパイプラインが**両方とも真**の場合のみ、&&リスト全体が**真**になるというものです。実際には左側のパイプラインから先に実行され、左側が偽であった場合は&&リスト全体が偽であることが確定してしまうため、右側のパイプラインは実行されません。この性質を利用して、&&リストを使って条件分岐を行うことが可能です。つまり、&&リストはif文の代わりに使うことができます。なお、この性質はC言語の&&演算子とも同じです。

if文で書き直す

冒頭の例をif文を使って書き直すと、リストAのようにになります。これらはどちらも同じ動作になります。なお、「if test -f file1」の部分はtestコマンドの別名を使って「if [-f file1]」と記述してもかまいません。

リストA &&リストをif文で書き直した例

```
if test -f file1 .....file1が存在すればtestコマンドが真になる
then .....真であればthenの後のリストが実行される
  cp file1 file2 .....file1をfile2にコピー
fi .....if文の終了
```

リストと、コマンドやパイプラインとの関係

一般に、コマンドは同時にパイプラインでもあり、リストでもあります。しかし、一般のリストはコマンドやパイプラインであるとは限りません。

たとえば、**リストB①**のls -lは、単純コマンドというコマンドであると同時にパイプラインであり、リストです。

一方、**リストB②**のcd /dir; lsという記述は、リストですが、これ全体はコマンドでもパイプラインでもありません。これを**リストB③**の{ cd /dir; ls; }のようにグループコマンドにすると、全体が1つのコマンド(複合コマンド)になり、同時にパイプラインでもリストでもあることになります。

リストA 終端されたリストと終端されていないリスト

```
if [ -f file1 ]; then ..... thenの前に改行がない場合;が必要
  cp file1 file2
fi
(cd /tmp; ls) .....のの前には;も改行もなくてもよい
```

リストB リストと、コマンド、パイプラインとの関係

```
ls -l .....①lsコマンドはそれ自体がパイプラインかつリスト
cd /dir; ls .....②これはリストだが、コマンドやパイプラインではない
{ cd /dir; ls; } .....③グループコマンドにするとコマンドやパイプラインにもなる
```

注意事項

空行の改行はよいが、空行の;はダメ

リストにおいて、改行と;とは基本的に同じ意味を持ちますが、改行は文脈によっては単なる区切り文字としての改行とみなされるため、一部;とは挙動が違って見える場合があります。

たとえば次の例のように、リストを記述していない空行の改行は、リストの区切りの改行ではなく、単なる区切り文字の改行とみなされるため、エラーにはなりません。しかし、空行の位置に;を記述すると、リストの区切りの;とみなされ、実際にはリストが存在しないため、文法エラーになります。

○正しい例

```
ls -l .....パイプライン+リストの区切りの改行
  .....単なる区切り文字の改行
cp file1 file2 .....パイプライン+リストの終端の改行
```

×誤った例

```
ls -l .....パイプライン+リストの区切りの改行
; .....リストの区切りの;とみなされ、リストがないのでエラーになる
cp file1 file2 .....パイプライン+リストの終端の改行
```

参照

&&リスト(p.37) ||リスト(p.39) if文(p.43) サブシェル(p.72) グループコマンド(p.74)

グループコマンドを使う

&&リストは、その左右にある、それぞれ1つのパイプラインにのみにかかります。これは、if文がパイプラインではなくリストにかかるのとは異なります。したがって、&&リストの右側で複数のパイプライン(たとえば複数のコマンド)を実行したい時は、**リストB**のようにグループコマンドの{ }を使って、パイプラインを1つのコマンドにまとめる必要があります。

リストB 右側のパイプラインをグループコマンドにした例

```
cmp -s file1 file2 && { .....file1とfile2の内容が同じなら真になる
echo '重複ファイルfile2を削除します' .....メッセージを表示
rm -f file2 .....重複したfile2を削除
} .....グループコマンドの終了
```

注意事項

if文との終了ステータスの相違点

&&リストの左側のパイプラインが偽の場合、&&リスト全体の終了ステータスは偽になりますが、それに相当するif文の終了ステータスは、if文の仕様により真になり、この部分については動作が異なることになります。

&&リストでの例

```
$ false && echo hello .....左側のパイプラインが偽の場合
$ echo $? .....その&&リストの終了ステータスは.....
1 .....偽になる
```

if文での例

```
$ if false; then echo hello; fi .....ifの直後のリストが偽の場合
$ echo $? .....そのif文の終了ステータスは.....
0 .....真になる
```

参照

if文(p.43)

グループコマンド(p.74)

||リスト



簡単な条件判断を行える

書式 **パイプライン1** || **パイプライン2**

例 `test -f file1 || exit 1`file1が存在しない場合はエラーで終了する

基本事項

||リストでは、まず**パイプライン1**が実行され、その結果が偽(終了ステータスが「0」以外)である場合のみ**パイプライン2**が実行されます。

終了ステータス

パイプライン1が偽でパイプライン2が実行された場合はパイプライン2の終了ステータス、パイプライン1が真の場合は「0」が、||リスト全体の終了ステータスになります。

解説

||リストの本来の意味は、||の左右のパイプラインの**どちらかが真ならば**||リスト全体が真になるというものです。実際には左側のパイプラインから先に実行され、左側が真であった場合は||リスト全体が真であることが確定してしまうため、右側のパイプラインは実行されません。この性質を利用して、||リストを使って条件分岐を行うことが可能です。つまり、||リストはif文の代わりに使うことができるのです。なお、この性質はC言語の||演算子とも同じです。

if文で書き直す

冒頭の例をif文を使って書き直すと、**リストA**のようになります。これらはどちらも同じ動作になります。||リストとif文とでは、条件判断の真偽が逆になるため、ここではtestコマンドに!という引数を付けて条件を反転していることに注意してください。なお、if test ! -f file1の部分はtestコマンドの別名を使ってif [! -f file1]と記述してもかまいません。

リストA ||リストをif文で書き直した例

```
if test ! -f file1 .....file1が存在しなければtestコマンドが真になる
then .....真であればthenの後のリストが実行される
    exit 1 .....エラーで終了
fi .....if文の終了
```

グループコマンドを使う

|| リストは、その左右にある、それぞれ1つのパイプラインにのみにかかります。これは、if文が、パイプラインではなくリストにかかるのとは異なります。したがって、|| リストの右側で複数のパイプライン(たとえば複数のコマンド)を実行したい時は、リストBのようにグループコマンドの{ }を使って、パイプラインを1つのコマンドにまとめる必要があります。

リストB 右側のパイプラインをグループコマンドにした例

```
test -f file1 || { .....file1が存在すれば真になる
echo 'file1が存在しません' .....偽の場合、エラーメッセージを表示
exit 1 .....エラーで終了
} .....グループコマンドの終了
```

参照

if文(p.43)

グループコマンド(p.74)

>第4章 複合コマンド

4.1 概要	42
4.2 構文	43
4.3 サブシェルとグループコマンド	72
4.4 シェル関数	76
4.5 算術式の評価と条件式の評価	80