図A evalを使った配列への代入と参照

```
$ week_0=Sunday week_1=Monday week_2=Tuesday ……week_0からweek_6までに値を代入
$ week_3=Wednesday week_4=Thursday
$ week_5=Friday week_6=Saturday
$ echo "$week_2" ……week[2]などはweek_2として参照すればよい
Tuesday ……正しく表示される
$ today=3 ……シェル変数todayに3を代入
$ eval echo \"\$week_$today\" ……❶week[today]はevalを使ってこのように参照する
Wednesday ……たしかにWednesdayと表示される
$ eval week_$today=\''W E D'\' ……❷week[today]='W E D'に相当する代入を実行
$ eval echo \"\$week_$today\" ……再びweek[today]を参照
W E D ……スペースも含め、正しく表示される
```

リストA for文を使った配列への一括代入

```
i=0 ……シェル変数iに0を代入
for day in Sunday Monday Tuesday Wednesday Thursday Friday Saturday……:
                                        配列に代入する値の分だけfor文でループ
do ……ループの開始
  eval week_$i='$day' ……week[i]=$dayに相当する代入を実行
  i=`expr "$i" + 1` ……シェル変数iに1を加える
done ……ループの終了
```

参照

eval(p.98)

>第14章 シェルスクリプトのノウハウ&定石

クォートのネスティングパズル

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

シェル上では、*、$、<、>、スペース、改行など、多数の記号が特殊な意味を持ちます。これらの特殊な意味を打ち消し、単なる文字として使用したい場合には**クォート**を用います。しかし、evalコマンドを使用した場合などのように、シェルによる解釈が2回以上行われることもあります。2回以上の解釈に耐えるクォートは、ある意味パズルのように複雑になります。本項では、複数回の解釈が行われても問題のないクォート方法や、その関連事項について解説します。

3回の解釈に耐えるためのクォート

表Aは、特殊な記号を含む文字列を、1〜3回まで解釈されても元の文字列になるように、クォートを付ける方法をまとめたものです。実際にシェル上で解釈させるために、echoコマンドとevalコマンドも、文字列の欄に含めて書いています。元の文字列には、$や*や[ ]などが含まれており、クォートがないとパラメータ展開やパス名展開が行われ、意図しない文字列に変わってしまいます。

まず、1回の解釈に耐えるためには、文字列全体をシングルクォート(' ')で囲めばOKです。表Aの1回解釈の例のようにechoコマンドを実行すれば、1回の解釈によってシングルクォートが取り除かれ、その中身の文字列がそのまま表示されます。なお、文字列中に普通の文字としてのシングルクォートがある場合については後述します。

2回の解釈に耐えるには、文字列全体をシングルクォート(' ')で囲んだものを、さらに\' \'で囲みます。表Aのようにeval echoを実行すると解釈が計2回行われ、1回目の解釈で内側の' 'が取り除かれると同時に外側の\' \'が単なる' 'に変わり、2回目の解釈でこの' 'が取り除かれて元の文字列になります。

さらに、3回の解釈に耐えるには、つまり表Aのようにeval eval echoに耐えるには、2回の解釈に耐える文字列の外側をさらに\\\' \\\'で囲みます。\\\'は、「\\」と「\'」を並べたものであり、1回解釈を行うとそれぞれ「\」と「'」になるため、結果として\'になります。よって、文字列全体は1回の解釈によって、表Aの2回解釈の文字列と同じものになるため、合計3回の解釈に耐えられるのです。

表A 解釈回数とそれに耐えるクォート

解釈回数	文字列の例
元の文字列	`$$ $012 *[Hello]* *[World]* $321 $?`
1回解釈	`echo '$$ $012 *[Hello]* *[World]* $321 $?'`
2回解釈	`eval echo \''$$ $012 *[Hello]* *[World]* $321 $?'\'`
3回解釈	`eval eval echo \\\'\''$$ $012 *[Hello]* *[World]* $321 $?'\'\\\'`

文字列中にシングルクォートがある場合

どのような文字列でも、基本的にはシングルクォート(' ')で囲んでおけば解釈は避けられます。ただし、シングルクォート自身は例外で、文字列中にシングルクォートがあると、このシングルクォートによってクォートが閉じてしまいます。

このような場合は、**リストA**のように、いったんシングルクォートを閉じてから、シングルクォート単体を\でクォートして並べ、その直後に再びシングルクォートで囲まれた文字列を続けます。これは、形式的に「シングルクォート中の'は'\''と記述する」と覚えておくとよいでしょう。

文字列中にパラメータ展開を含ませたい場合

すでにシングルクォートでクォートしている文字列中で、一部のみシェル変数の参照などのパラメータ展開を行わせたいことがあります。このような場合、簡単な方法としてはシングルクォートをダブルクォートに変更すればよいでしょう。しかし、全体をダブルクォートに変更したのでは、パラメータ展開ではない$や、コマンド置換の`が解釈されてしまう可能性があります。

そこで、**リストB**のように、本当にパラメータ展開したい部分のみを" "で囲み、それ以外の部分は元通り' 'で囲んだままにするのがよいでしょう。つまり、シングルクォートの途中でクォートをいったん閉じ、その直後にダブルクォートで囲まれたパラメータ展開を続け、その直後に再びシングルクォートを続けます。これは、形式的には「シングルクォート中のパラメータ展開は'"$パラメータ"'と記述する」と覚えるとよいでしょう。

リストA シングルクォートを含む文字列のクォート方法

```
echo '<< I don'\''t know. >>'  ……いったんシングルクォートを閉じて\'を挿入
```

リストB 文字列中でパラメータ展開する方法

```
echo '< hello '"$USER"' hello >'  ……いったんシングルクォートを閉じて"$USER"を挿入
```

参照

eval(p.98)　シングルクォート' '(p.207)　バックスラッシュ\(p.211)
ダブルクォート" "(p.209)

if文の代わりに && や||を使う

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

簡単なif文の構造は、if文を使わずに&&リストや||リストで表現できます。たとえば、以下のような記述です。

❶`cmp -s file1 file2 && ln -sf file1 file2` ……file1とfile2の内容が同じなら、シンボリックリンクにする

❷`[ -f file ] || exit 1` ……fileが存在しなければエラーで終了する

ここで&&リストや||リストの右側のリストにグループコマンドの{ }を使えば、条件成立時に複数のコマンドを実行することができます。

さらに、次のように工夫することによって、if文のelseの構造を表現することも可能です。

elseを含むif文の構造を表現

&&リストと||リストを使ってelseを含むif文の構造を表現した記述例を**リストA**に示します。このように、testコマンドなどの条件判断のためのコマンドとグループコマンドの{ }とを&&リストでつなぎ、{ }の中に条件が「真」の場合に実行するコマンドを記述します。この時、{ }内の最後のコマンドとして必ず:コマンドを実行して終了ステータスが「真」になるようにします。

そのあと、||リストで別の{ }とつなげば、この{ }がelseの条件で実行されるようになります。なお、上記の:コマンドがないと、testコマンドの結果が真の場合に実行される「command_a」の終了ステータスが偽だった場合に、誤ってelseの部分の{ }まで実行されてしまうため、注意してください。

リストA elseを含むif文の構造を表現

```
[ "$i" -lt 10 ] && {          ……testコマンドの結果が真ならば&&リストの右側の{ }を実行
  echo 'iの値は10未満です'    ……条件が真の場合のメッセージを表示
  command_a                   ……ここに実行したいコマンドを書く
  :                           ……終了ステータスを真にするため、{ }の最後に:コマンドを実行しておく
} || {                        ……||リストを使ってelseの意味を表現
  echo 'iの値は10以上です'    ……最初の条件が偽の場合のメッセージを表示
  command_b                   ……ここに実行したいコマンドを書く
}                             ……elseのグループコマンドの終了
```

参照

&&リスト(p.37)　||リスト(p.39)　グループコマンド(p.74)　:コマンド(p.87)

do〜while構造の実現

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

シェルスクリプトでループを実現する構文にはfor文とwhile文がありますが、このうち、一定の条件が満たされているかぎりループするのはwhile文です。while文の条件チェックは、1回目のループよりも前に行われるため、while文の開始直前にループの条件が満たされていなかった場合は、ループは1回も実行されません。

しかし、場合によっては「まず1回ループの中身を実行し、その結果によってループを継続するかどうか判断したい」こともあります。このような構文は、C言語ではdo〜while文として存在します。シェルスクリプトでも、次のような方法でdo〜while文と同様の構造を実現できます。

通常のwhile文

まずは普通のwhile文の例です。**リストA**は、シェル変数iに保持されている「1、2、3…」と順に増加する整数を、シェル変数sumにどんどん加算して行き、sumの値がはじめて「100」以上になった時にループを終了するというものです。実際に実行すると「i=14」「sum=105」になるはずです。

このwhile文では、はじめに「sum=0」と初期化しているにもかかわらず、その直後にwhile文のtestコマンドによってsumが「100」未満かどうかのチェックが行われます。これは真であることが明らかであるため、無駄なチェックをしていることになります。

リストA 通常のwhile文での記述

```
i=0                           ……シェル変数iの値を0に初期化
sum=0                         ……シェル変数sumの値を0に初期化
while [ "$sum" -lt 100 ]      ……sumの値が100未満であるかぎりループ
do                            ……ループの開始
  i=`expr "$i" + 1`           ……iの値に1を加算する
  sum=`expr "$sum" + "$i"`    ……sumの値にiの値を加算する
done                          ……ループの終了

echo 'i='"$i"                 ……シェル変数iの値を表示
echo 'sum='"$sum"             ……シェル変数sumの値を表示
```

do～while構造を記述したwhile文

リストAをdo～while構造を記述したwhile文で書き直すと**リストB**のようになります。このように、while文の文法では、whileの直後も、doの直後も、記述するのは「リスト」であり、リストは複数のパイプラインの集まりであるため、結局whileの後に複数のコマンドを記述できます。すなわち、実質的に「while～doの間でループを記述してもかまわない」のです。この場合、リストの終了ステータスはリスト中の最後のパイプライン、つまり「最後のコマンドの終了ステータス」になるため、結局、最後に記述したtestコマンドの終了ステータスによってループを続行するかどうかが判断されます。また、本来のwhile文でループを記述するべきdo～doneの間は、何も実行する必要がないため、:コマンドを記述するのみにします。

このようにdo～while構造を実現したリストBでは「sum=0」と代入したあと、リストAとは違って条件がチェックされずに1回目のループ内容が実行され、最後にtestコマンドでループ条件を判断するため、無駄がなくなります。

リストB do～while構造を記述したwhile文

```
i=0 ……シェル変数iの値を0に初期化
sum=0 ……シェル変数sumの値を0に初期化
while ……do～while風ループの開始
  i=`expr "$i" + 1` ……iの値に1を加算する
  sum=`expr "$sum" + "$i"` ……sumの値にiの値を加算する
  [ "$sum" -lt 100 ] ……sumの値が100未満であるかぎりループ
do :; done ……do～while風ループの終了
                                (本来のループ内は:コマンド)
echo 'i='"$i" ……シェル変数iの値を表示
echo 'sum='"$sum" ……シェル変数sumの値を表示
```

参照

while文(p.63)　　:コマンド(p.87)

すべての引数についてループする

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

シェルスクリプトの起動時に、シェルスクリプトに付けられていた各引数すべてについてループしながら一定の処理を行いたいことがよくあります。本項では、すべての引数についてループする方法について解説します。

for文を使う方法

すべての引数についてループするには、for文を使うのが基本です。**リストA**のように「for 変数名」の後に「in "$@"」と書けば、すべての位置パラメータがそのままfor文で使用されます(set -uの環境も考慮する場合は「in ${1+"$@"}」または「in ${@+"$@"}」とします)。なお、「in "$@"」は省略可能で、単に「for 変数名」だけを記述してもかまいません。

ループ中では、現在の引数が指定したシェル変数であるargに入っているため、ループの中で"$arg"を使って目的のコマンドを記述します。

for文を使う方法は単純明快ですが、ループ中でさらに次の引数を参照することができません(その場合は後述のwhile文を使う必要があります)。なお、for文によるループでは、ループ終了後も位置パラメータの値はそのままの状態で残ります。

while文を使う方法

すべての引数についてwhile文でループするには、**リストB**のようにします。while文では、位置パラメータの個数の入った特殊パラメータ$#を参照して、この値が「0」より大きいかぎりループを行うようにします。while文の最後では必ずshiftコマンドを実行し、次のループのために位置パラメータをシフトするようにします。

ループ中では、現在の引数が"$1"に入っているため、"$1"を参照して、目的のコマンドを実行します。もし、現在の引数の次の引数も同時に参照したい場合は、"$2"以降を参照することができます。この場合、すでに解釈した引数は適宜shiftして、次のループに進むようにします。

このように、while文を使う方法は自分で位置パラメータをshiftする必要があり、やや記述内容が増えますが、「次の引数を参照することもできる」などfor文ではできない処理も行えます。なお、while文のループ終了時には、すべての位置パラメータがシフトされ、位置パラメータは未設定の状態になります。

リストA すべての引数についてループ(for文)

```
for arg in "$@" ……シェル変数argに位置パラメータを順に代入してループ
do ……ループの開始
  echo "$arg" ……試しに現在の引数の値を表示
  : ……ここに実行したいコマンドを書く
done ……ループの終了
```

リストB すべての引数についてループ(while文)

```
while [ $# -gt 0 ] ……位置パラメータの個数が0より大きいかぎりループ
do ……ループの開始
  echo "$1" ……試しに現在の引数の値を表示
  : ……ここに実行したいコマンドを書く
  shift ……位置パラメータをシフトする
done ……ループの終了
```

引数中のオプションを認識する方法

すべての引数について単純にループするだけでなく、-vや-o *file* といったオプションを認識してから残りの引数についてループしたい場合は、**リストC**のようにします。

ここでは、-vという単独のオプションと、-o *file*という後続の引数をとるオプションを認識し、それぞれシェル変数verboseとoutfileに結果を代入するようにしています。

この、オプションの認識部分は、リストのようにwhile文によるループと、ループ中のcase文によって行います。ここで、-oオプションの場合、次の引数を"$2"で参照した上でshiftを実行している点に注目してください。なお、--というオプションが現れると、オプションの終了とみなしてwhile文をbreakするようにしています。

while文が終了した状態では、シェルスクリプトの引数のうち、オプションに相当する部分がすでにシフトされてなくなっています。したがってこのあと、前述のfor文によるループと同じことをやれば、残りの引数についてループして、目的のコマンドを実行できます。

なお、オプションの認識部分については、getoptsコマンドを使って行うこともできます。

リストC 引数中のオプションを認識して、残りの引数についてループ

```
verbose=0 ……シェル変数verboseを0に初期化
outfile=default ……シェル変数outfileをdefaultに初期化

while [ $# -gt 0 ] ……位置パラメータの個数が0より大きいかぎりループ
do ……ループの開始
  case $1 in ……オプションの種類によって分岐
    -v) ……-vオプションだった場合
      verbose=1 ……verboseに1を代入(-vオプションありと記憶)
      ;; ……このパターンの終了
    -o) ……-oオプションだった場合
      outfile=$2 ……-oの次の引数をoutfileに代入
      shift ……次の引数をすでに読んだので、引数1個分シフト
      ;; ……このパターンの終了
    --) ……--オプションだった場合
      shift ……この引数をシフトして無視する
      break ……オプションの終了と解釈し、whileループを抜ける
      ;; ……このパターンの終了
    *) ……それ以外の引数だった場合
      break ……オプションの終了と解釈し、whileループを抜ける
      ;; ……このパターンの終了
  esac ……case文の終了
  shift ……次のループのため引数をシフトする
done ……ループの終了

echo verbose="$verbose" ……確認のため、verboseの値を表示
echo outfile="$outfile" ……確認のため、outfileの値を表示

for arg in "$@" ……オプション以外の残りの引数についてfor文でループする
do ……ループの開始
  echo "$arg" ……確認のため、引数を表示
  : ……ここに実行したいコマンドを書く
done ……ループの終了
```

参照

for文(p.55) while文(p.63) case文(p.49) getopts(p.108)

一定回数ループする

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

シェルスクリプトで、ある処理を一定回数繰り返したい、といった場面はよくありますが、一定回数の繰り返しを行うには次のような記述方法があります。

for文で数値を羅列する方法

繰り返し回数が比較的少ない場合、**リストA**のように、for文を使い、適当な数値を繰り返し回数分だけ羅列してしまうのがよいでしょう。この方法では、ループ変数用のシェル変数の加算のためにexprコマンドを呼び出す必要がないため、高速に動作します。また、羅列するのは数値でなくてもよく、適当な文字列を必要なループ回数分だけ記述してもかまいません。

数値を羅列したfor文のネスティング

さらに、数値を羅列したfor文をネスティングすることにより、100回程度の繰り返し回数でも記述できます。**リストB**は、0～99までの100回の繰り返しを行う例です。ここでは、シェル変数iが10の位を、シェル変数jが1の位を担当し、"ij"で2桁の値になります。シェル変数iの最初の値は空文字列(for文で''と記述)で、これにより、"ij"の値が0～9となる1桁の値も表現しています。

リストA for文で数値を羅列して一定回数ループ

```
for i in 1 2 3 4 5 6 7 8 ……8回ループのため、数値を直接羅列
do ……ループの開始
  echo "$i" ……試しにシェル変数iの値を表示
  : ……ここにループで実行したいコマンドを記述する
done ……ループの終了
```

リストB for文のネスティングで100回のループ

```
for i in '' 1 2 3 4 5 6 7 8 9 ……10の位の数値を羅列（最初は空文字列）
do ……10の位のループの開始
  for j in 0 1 2 3 4 5 6 7 8 9 ……1の位の数値を羅列
  do ……1の位のループの開始
    echo "$i$j" ……試しに"$i$j"の値（0～99）を表示
    : ……ここにループで実行したいコマンドを記述する
  done ……1の位のループの終了
done ……10の位のループの終了
```


while文を使った一定回数のループ

前述のように数値を羅列するのではなく、普通にループを記述するには、**リストC**のようにwhile文を使います。この場合はループ変数として用いているシェル変数に「1」を加えるためにexprコマンドを呼び出す必要があるため、動作は遅くなります。

リストC while文とexprで一定回数のループ

```
i=0 ……シェル変数iの値を0に初期化
while [ "$i" -lt 100 ] ……iの値が100未満であればループ
do ……ループの開始
  echo "$i" ……試しにiの値を表示
  : ……ここにループで実行したいコマンドを記述する
  i=`expr "$i" + 1` ……iの値に1を加算する
done ……ループの終了
```

シェル変数に1を加算する方法

シェル変数に「1」を加算するには、**表A**のようにexprコマンド以外にもいくつかの方法が存在します。この表Aのexpr以外の記述を、リストCのexprコマンドの行の代わりに使用できます。算術式展開や算術式の評価はシェル内部で実行されるため、外部コマンドのexprを使うよりも高速です。したがって、bashやFreeBSDのshではこれらを利用してもよいでしょう。ただし、移植性を重視する場合はやはりexprコマンドを使う必要があります。

表A シェル変数iに1を加算する方法※

記述	意味	Linux (bash)	FreeBSD (sh)	Solairs (sh)
i=`expr "$i" + 1`	exprをコマンド置換で取り込み	○	○	○
i=$(($i + 1))	算術式展開(シェル変数は$iで参照)	○	○	×
i=$((i + 1))	算術式展開(シェル変数はiで参照)	○	○	×
i=$[i + 1]	算術式展開の別の記述法	○	×	×
((i = i + 1))	算術式の評価で普通に加算と代入を行う	○	×	×
((i += 1))	算術式の評価で+=演算子で加算と代入を行う	○	×	×
((i++))	算術式の評価でインクリメント	○	×	×

※ 本表のほかにletコマンドを使う方法もあるが、letはbashとFreeBSDのshで仕様が異なるため、割愛する

連番のブレース展開を使ったループ

bash 3以降では、{1..9}の形式の連番のブレース展開が使えるため、**リストD**のようにブレース展開を使ってループを記述することもできます。数字の代わりに{a..z}と記述すれば、aからzまでのアルファベット26文字でループすることができます。

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

リストD 算術式のfor文を使ったループ

```
for i in {0..99} ……連番のブレース展開を使って0から99までループ
do ……ループの開始
  echo "$i" ……試しにiの値を表示
  : ……ここにループで実行したいコマンドを記述する
done ……ループの終了
```

算術式のfor文を使ったループ

bash限定なら、**リストE**のように算術式のfor文を使ってループを記述することもできます。移植性は低いものの、記述形式がC言語に近く、ある意味合理的といえるかもしれません。

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

リストE 算術式のfor文を使ったループ

```
for ((i = 0; i < 100; i++)) { ……シェル変数iを使って0から99までループ
  echo "$i" ……試しにiの値を表示
  : ……ここにループで実行したいコマンドを記述する
} ……ループの終了
```

Memo

●数字の羅列は、seqコマンドを使って得ることもできます。seqは連番を標準出力に出力するコマンドです。たとえば1～100までの数値でループしたい場合、次のように書けます。

```
for i in `seq 1 100`
do
  echo "$i"
  :
done
```

参照

for文(p.55)　while文(p.63)　expr(p.261)　算術式展開(p.232)
算術式の評価(( ))(p.80)　算術式のfor文(p.61)

ラッパースクリプト

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

すでにインストールされている何らかのコマンドに対し「環境変数の設定」や「常に使用するオプションの付加」など、前処理を行った上でコマンドを起動するようにしたシェルスクリプトのことを、**ラッパースクリプト**(*wrapper script*)といいます。

ラッパースクリプトを使えば、コマンドのちょっとした修正についてはそのコマンド本体を再コンパイルすることなく、シェルスクリプト上の前処理によって行うことができます。

ラッパースクリプトの例

ラッパースクリプトの例を**リストA**に示します。ここでは、**LANG**と**MY_ENV**という環境変数を設定した上で、さらに-myoptionというオプション引数を常に付加した上でmycommandというコマンドを実行しています。

このラッパースクリプト起動時の引数については、"$@"によってそのままmycommandに引き渡されます。このように、シェルによる解釈を一切行わずに引数を引き渡すには"$@"を使うのが定石です。間違っても"$*"やクォートなしの$@を使用してはいけません[注1]。

また、mycommandはexecによって起動しているため、このシェルスクリプトのプロセスIDのままでmycommandが起動され、プロセスが無駄になりません。ちなみにexecを省略しても一応動作はしますが、その場合はmycommandの実行後も、シェルスクリプトのプロセスがmycommandの終了まで残ったままになってしまい、プロセスが無駄になります。ラッパースクリプトでは、最後に起動するコマンドにはexecを使うのが定石です。

リストA 環境変数の設定とオプションの追加をあらかじめ処理する

```
#!/bin/sh

LANG=C; export LANG ……環境変数LANGをCに設定
MY_ENV=my_env; export MY_ENV ……環境変数MY_ENVをmy_envに設定
exec mycommand -myoption "$@" ……mycommandに-myoptionを追加してexecで起動
```

参照

特殊パラメータ"$@"(p.167)　exec(p.100)

注1　詳しくは「特殊パラメータ　"$@"」(p.167)を参照してください。

ファイル名に日付文字列を含ませる

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

シェルスクリプト中で、コマンドの実行結果などをファイルに保存する際に、そのファイル名に日付の文字列を含ませたいことがあります。このような場合には、日付や時刻を表示するdateコマンドの出力をコマンド置換で取り込み、ファイル名の一部として利用します[注2]。

日付を含むファイル名のファイルを作成

dateコマンドは、普通に実行すると単に日付と時刻を表示しますが、**リストA**のように、dateコマンドの引数に+で始まる書式指定文字列を付ければ、その日付と時刻の表示書式を指定することができます。おもな書式指定文字列を**表A**に示します。

リストAでは、date +%Y%m%d-%H%M%Sと書式指定されているため、これを仮に「2038年1月19日の12時14分7秒」に実行したとすると、「20380119-121407」という文字列が出力されます。この文字列をコマンド置換で取り込み、その頭にlog-を付けているため、結局リストAでは、mycommandの標準出力が「log-20380119-121407」というファイルに書き込まれることになります。

なお、dateコマンドには、表A以外にも書式指定文字列が存在しますが、使用できる書式文字列の種類は、そのOSにインストールされているdateコマンドのバージョンによって若干違いがあるため、詳しくはオンラインマニュアルを参照してください。ただし、少なくとも表Aの書式は、Linux／FreeBSD／Solarisで共通して使用できます。また、書式によっては、日本語環境で実行すると日本語の日付文字列が表示され、ファイル名としては不適切になる場合がありますが、表Aの書式は日本語環境でもすべて数字のみの表示となるため、安心して使えます。

リストA　日付を含むファイル名のファイルを出力

```
mycommand > log-"`date +%Y%m%d-%H%M%S`"  ……mycommandの標準出力を、
                                          日付を含むファイル名のファイルに出力
```

表A　dateコマンドのおもな書式

書式	表示内容
%Y	年（1970～2000～の西暦4桁表示）
%m	月（01～12の2桁表示）
%d	日（01～31の2桁表示）
%H	時（00～23の2桁24時制表示）
%M	分（00～59の2桁表示）
%S	秒（00～59の2桁表示）※

※　うるう秒の場合「60」以上の表示になる場合もある

注2　コマンド置換については9.3節を参照してください。

ファイルを1行ずつ読んでループする

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

ファイルから1行ずつ読み込み、その内容を解釈しつつループするには、「while read」の形式を使います。readコマンドは標準入力から1行ずつシェル変数に読み込み、入力がEOFになると偽(1)の終了ステータスを返します。よって、「while read」の形でループを記述すれば、1行ごとにファイル全体にわたってループすることができます。

readコマンドの引数に複数のシェル変数を指定すると、シェル変数**IFS**の値を区切り文字として単語分割された結果が複数のシェル変数に分割して代入されるため、これを利用して簡単な字句解析ができます。

/etc/hostsを読んで、IPアドレスに対応するホスト名を表示

リストAは、/etc/hostsを1行ずつ読み込みながらループし、シェルスクリプトの引数で指定されたIPアドレスに一致する行を見つけたら、そのホスト名部分を表示して終了するというシェルスクリプトです。

ここでは、あらかじめリストA❶のようにexecコマンドを使ってファイル記述子「3」に/etc/hostsを読み出し用としてオープンしておきます。**リストA❷**のreadコマンドではファイル記述子のリダイレクトを使ってファイル記述子「3」から読み込みます。readコマンドで読み込むためのシェル変数は、左からip、hostと指定してあるため、シェル変数ipには/etc/hostsファイルのIPアドレスの文字列が、シェル変数hostにはホスト名部分の文字列が読み込まれるはずです。

whileループ中では、シェルスクリプトの引数$1と、シェル変数ipが一致するかどうかをcase文を使って判定し、一致した場合はシェル変数hostの内容を表示して、そのままexit 0を実行します。つまり、IPアドレスが見つかった場合はwhile文の途中でシェルスクリプトを終了することになります。

一方、whileループを最後まで終了した場合は、一致するIPアドレスが見つからなかったということなので、その旨をエラーメッセージで表示して、exit 1で終了しています。なお、whileループを終了直後、**リストA❸**で念のためファイル記述子「3」をクローズしていますが、これは省略してもかまいません。

ところで、この例のように/etc/hostsをあらかじめ「exec 3<」でオープンしてから使用しているのは、Solarisのshを含めて正常動作するようにするためです。bashやFreeBSDのshの場合は、後述のようにwhile文全体に/etc/hostsを単純にリダイレクトしてもかまいません。

リストA /etc/hostsを読んで、IPアドレスに対応するホスト名を表示

```
case $# in ……引数の個数をチェック
  1) ……引数が1個ならOK
    ;;
  *) ……引数が1個以外の場合は、
    echo "Usage: $0 ip-address" 1>&2 ……Usage:のメッセージを表示して、
    exit 1 ……エラーで終了
    ;;
esac

exec 3< /etc/hosts ……ファイル記述子「3」を使って/etc/hostsを読み出しオープン❶

while read ip host 0<&3 …… ファイル記述子 「3」 から1行ずつ読み込んでループ❷
do
  case $1 in …… シェルスクリプトの引数が、
    $ip) …… IPアドレスの部分に一致した場合は、
      echo "$host" …… ホスト名の部分を表示
      exit 0 …… ループを中断し、正常終了
      ;;
  esac
done

exec 3<&- …… 念のためファイル記述子「3」をクローズ(省略可能)❸

echo "$1 not found" 1>&2 …… 一致するIPアドレスが見つからなかった旨を表示
exit 1 …… エラーで終了
```


while文全体にファイルを直接リダイレクトする方法

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

bashまたはFreeBSDのshでは、前述のリストAはリストBのように記述することができます。ここでは、「exec 3< /etc/hosts」は実行せず、while文全体に/etc/hostsをリダイレクトします。while文に対するリダイレクトは、最後のdoneの行に「done < /etc/hosts」のように記述します。

リストBでは、while文の標準入力が/etc/hostsになっているため、readコマンドではそのまま標準入力を読めば/etc/hostsが読めます。その他の処理は前述リストAと同じです。

ところで、リストBをSolarisのshで実行すると、一致するIPアドレスが見つかった場合でもwhile文の途中のexitでシェルスクリプトを終了することができず、whileループを抜けた後でIPアドレスが見つからないというエラーメッセージが表示されてしまいます。これは、Solarisのshでは、while文に対してリダイレクトを行うと、while文全体が暗黙のサブシェルになってしまうのが原因です。サブシェルの中でのexitはサブシェル(つまりwhile文)を終了するだけで、シェルスクリプトを終了できません。

リストB while文全体にファイルを直接リダイレクトする方法

```
case $# in ……引数の個数をチェック
  1) ……引数が1個ならOK
    ;;
  *) ……引数が1個以外の場合は、
    echo "Usage: $0 ip-address" 1>&2 ……Usage:のメッセージを表示して、
    exit 1 ……エラーで終了
    ;;
esac

while read ip host ……標準入力から1行ずつ読み込んでループ
do
  case $1 in ……シェルスクリプトの引数が、
    $ip) ……IPアドレスの部分に一致した場合は、
      echo "$host" ……ホスト名の部分を表示
      exit 0 ……ループを中断し、正常終了
      ;;
  esac
done < /etc/hosts ……while文全体の標準入力を/etc/hostsにリダイレクト

echo "$1 not found" 1>&2 ……一致するIPアドレスが見つからなかった旨を表示
exit 1 ……エラーで終了
```

Memo

- bashまたはFreeBSDのshでも、while文をパイプに接続した場合、つまり「`cat file | while read …`」の形で記述するとwhile文が暗黙のサブシェルになるため、whileループ中で`exit`できなかったり、設定したシェル変数等をwhileループを抜けた後に持ち越せないなどの問題が発生します。シェルとしてkshまたはzshを使うと、この場合でもwhile文はサブシェルになりません。

>Appendix
サンプルスクリプト