2016 年 05 月 13 日

計算機実習 III (2016 年度)
第 4 回: コマンドプロンプトとバッチファイル その 4

(http://takeno.iee.niit.ac.jp/%7Eshige/
math/lecture/comp4/comp4.html)

目 次

1 対話型での変数値の取得

バッチファイルの内部変数値をバッチファイルの外から変えるには、以下のような方法がある。

1. バッチファイルの実行前に環境変数を変更しておく
2. set /p でバッチファイル実行中に対話的に変更する
3. バッチファイルにオプションをつけて実行する

1. は、バッチファイル内ではその環境変数の初期値を設定せず、バッチファイルを実行する前に set でその初期値を設定する、という方法である。この節では 2. の方法を説明し、3. の方法は次節で紹介する。

コマンド	意味
set /p [変数名]=[文字列]	[文字列] を表示して入力待ちになり、入力したものを [変数名] の変数に代入して次へ進む

表 1: set /p による変数の取得

set コマンドに「/p」オプションをつけてを使用すると、C 言語の scanf() のようにバッチファイルの実行中に対話的に変数値を入力することができる (表 1)。

なお、set /p の = の右辺に書く文字列は、変数に代入される文字列「ではなく」あくまで画面表示用の文字列であり、実際に左辺の変数に代入されるのは、対話的に入力される文字列 (Enter の手前までの文字列) であることに注意せよ。例:

```
@echo off
set /p x=3桁の整数を入力してください:
set /a x1=x/100
set /a x2=(x%%100)/10
set /a x3=x%%10
echo %x3%%x2%%x1%
```

このバッチファイルを実行すると、2 行目の「set /p」で = の後ろの文字列「3桁の整数を入力してください:」を表示して入力待ちになり、その後にキーボードから入力された文字列が変数 x に代入される。3 行目から 5 行目は、その x の各桁の数を一つずつ取り出す計算で、最後にそれを逆順にして表示している。

2 バッチファイルへのオプション

次に、前節 3. のバッチファイルへのオプションを使用する方法を説明する。

バッチファイルにオプションをつけて実行すると、そのオプション文字列はバッチファイル内で表 2 のような名前で変数値として利用できる。

名前	意味
%1,%2,...,%9	指定した順の個々のオプション文字列
%0	そのバッチファイル自身の名前
%*	%1 以降のオプション文字列全体

表 2: バッチファイルのオプション値の参照

例えば

```
Z:¥> test1.bat 123 竹の 茂治
```

のように実行すると、このバッチファイル内では

```
%1=123, %2=竹の, %3=茂治, %0=test1.bat,
%*=123 竹の 茂治
```

となる。また、%4 以降はこの場合は空文字列となる。例:

```
@echo off
set /a x=%1 + %2
echo %1 たす %2 は %x% です。
```

のバッチファイル add.bat にオプションをつけてコマンドプロンプトで実行すると、例えば以下のようになる。

```
Z:¥> add.bat 35 29
35 たす 29 は 64 です。
```

この 2 行目が、バッチファイルの 3 行目の echo コマンドによって出力される行である。

注意:

- バッチファイルのオプションは、一度に参照できるのは %1 から %9 までの 9 個であり、%10 という変数を「直接」使うことはできず、%10 は %1 の値に文字 0 がついた文字列とみなされる。

バッチファイルの 10 番目以降のオプションを参照したり、すべてのオプションに渡る処理を順にしたい場合には shift コマンドを用いる方法がある (表 3)。

コマンド	意味
shift	%[n] をそれぞれ %[n-1] にシフトする
shift /[番号]	指定番号以降のみシフトする

表 3: shift の使用法

例えば、

```
Z:¥> test1.bat 1 2 3 4 5 6 7 8 9 10 11 12
```

とすると、

```
%1=1, %2=2, ..., %9=9, %0=test1.bat,
%*=1 2 3 4 5 6 7 8 9 10 11 12
```

となるが、test.bat の中で 1 回 shift を実行すると

```
%1=2, %2=3, ..., %9=10, %0=1,
%*=1 2 3 4 5 6 7 8 9 10 11 12
```

となり、もう 1 回 shift を実行すると

```
%1=3, %2=4, ...%9=11, %0=2,
%*=1 2 3 4 5 6 7 8 9 10 11 12
```

となる。この例のように、引数全体を表す文字列 %* は、shift をしても変わらない。

また、元々はバッチファイル名だった %0 も、shift によって %1 だったものに変わるが、%0 を変更したくなければ「shift /1」とすればよい。例えば、test1.bat が

```
@echo off
echo [0]=%0, [1]=%1, [9]=%9, [*]=%*
shift /1
echo [0]=%0, [1]=%1, [9]=%9, [*]=%*
shift
echo [0]=%0, [1]=%1, [9]=%9, [*]=%*
```

であるとき、

```
Z:¥> test1.bat 1 2 3 4 5 6 7 8 9 10 11 12
```

とすると、

```
[0]=test1.bat, [1]=1, [9]=9, [*]=1 2 3 4 5 6 7 8 9 10 11 12
[0]=test1.bat, [1]=2, [9]=10, [*]=1 2 3 4 5 6 7 8 9 10 11 12
[0]=2, [1]=3, [9]=11, [*]=1 2 3 4 5 6 7 8 9 10 11 12
```

と表示され、最初の「shift /1」では%0 は変更されずにその後ろがシフトされるのに対し、2 度目の「shift」では %0 も含めて シフトされていることがわかる。

なお、この shift により逆に %0 や %1 だったものが失われてしまうが、これを復帰する専用のコマンドはない。

3 リダイレクション

コマンドプロンプト内でのコマンドの画面出力は、**標準出力** (C 言語の stdout) と呼ばれる仮想デバイスを介して行われている。C 言語で言えば、printf() や putchar() 等の関数は標準出力へ出力をする関数である:

コマンド（や printf() など） $\xrightarrow{(1)}$ 標準出力 (stdout) $\xrightarrow{(2)}$ 画面

コマンドプロンプトやバッチファイルでは、この標準出力と画面とのつながり (2) を、表 4 の 2 種類の **出力リダイレクション** を使うことで画面からファイルにつなぎかえることができる。

種類	名前
[コマンド] > [ファイル]	出力リダイレクション (上書き形式)
[コマンド] >> [ファイル]	出力リダイレクション (追加出力形式)

表 4: 出力リダイレクション

例えば

```
Z:¥> time /t > time1.dat
```

とすると、本来画面に表示される「time/t」の出力 (現在の時刻) が、ファイル time1.dat に保存される。

出力リダイレクションの「>」と「>>」の違いは以下の通り。

- 「[コマンド] > [ファイル]」は [ファイル] がなければ新規に作成し、[ファイル] があれば上書きする (削除して新規に作るのと同等)。
- 「[コマンド] >> [ファイル]」は [ファイル] がなければ新規に作成し、[ファイル] があればその末尾に追加出力する。

例えば、上の時間の保存を

```
Z:¥> echo %time% >> time1.dat
```

のような追加出力にすれば、ログイン時刻の記録を残すことなどにも使える。

数行程度の簡単なテキストファイルであれば、エディタを使わなくても echo コマンドと出力リダイレクション「>, >>」を使うだけで作ることができる。例:

```
Z:¥> echo ファイルの 1 行目 > test.dat
Z:¥> echo ファイルの 2 行目 >> test.dat
Z:¥> echo ファイルの 3 行目 >> test.dat
```

上の 1 行目で「>」を使うのは、test.dat というファイルが既にあったときでもその中身を消して新たに書き始めるためで、2,3 行目で「>>」を使うのは、「>」では最後に実行した echo の行 1 行しかファイルに残らないためである。

空デバイス nul (または NUL) を出力リダイレクション先に指定すると、結果はどこにも出力されないので単にコマンドの画面出力を消すのに使うことができる。例えば

```
@echo off
echo 適当に打たんかい
pause > nul
```

というバッチファイルを実行すると、3 行目の pause コマンドの出力である「続行するには何かキーを押してください...」が画面に出力されず、2 行目の「適当に打たんかい」が表示された後で入力待ちとなる。このようにすれば、pause のデフォルトのメッセージを削除/変更できる。

なお、2 行目と 3 行目の順番を逆にすると、何も表示されずに入力待ちになり、Enter 入力後に「適当に打たんかい」が表示されてしまうことになる。

4 ワイルドカード

複数のファイルを 1 度に指定する方法としてワイルドカードがある。**ワイルドカード**は、「?」と「*」の文字であるが、パスやファイル名の一部としてこれらを使った場合、その部分を任意の文字や文字列と見て、それにマッチ (適合) するすべてのファイルを並べて指定したことにする、といった仕組みになっている (表 5)。

記号	意味
?	0 文字か任意の 1 文字にマッチする
*	任意の長さの任意の文字列にマッチする

表 5: ワイルドカード

例えば、カレントディレクトリに

```
test.bat, test2.c, test2.exe, test3.bat, test4
```

のファイルやディレクトリがある場合、

```
test?.bat  =  「test.bat test3.bat」
test2.*    =  「test2.c test2.exe」
test?.*    =  上の 5 つのファイル名すべてを並べた文字列
```

となる。なお、拡張子のない「test4」という名前は、「test4.」と同じとみなされることになっているので、最後のパターンには test4 もマッチする。例:

```
Z:¥> copy test2.* dir
```

とすると、これは

```
Z:¥> copy test2.c test2.exe dir
```

と実行したことと同じことになる。

1 文字だけの * はすべてのファイル名にマッチし、???.??? は [名前].[拡張子] の形式のファイル名で、[名前] の部分が 3 文字以下、[拡張子] の部分が 3 文字以下のファイル名にマッチする。

コラム: テキストファイルとバイナリファイル

「メモ帳」で編集できるような、可読文字がただ並んでいるファイルを**テキストファイル** と呼び、それ以外のものを **バイナリファイル** と呼ぶ。C のコンパイル前のプログラムファイル (ソースファイルという) はテキストファイルであるが、ワープロファイル (.doc) や表計算データ (.xls) はバイナリファイルである。「バイナリ」とは元々は「2 進数」という意味だが、コンピュータにしか理解できない 2 進数が並んでいる、というところからの命名だろうか。

テキストファイルもバイナリファイルも、バイト (0x00–0xff) のデータが並んでいるだけ、ということでは同じだが、ASCII コードでの可読文字である 0x20–0x7e の英数半角記号、および改行記号 (0x0d, 0x0a) 等の印字用制御文字だけで構成されるものを狭義にはテキストファイルと呼ぶ。しかしこれだと日本語文字が含まれないので、日本語コード (通常 2 バイトコード) など、ある特定の文字コードが入ったものも広義にはテキストファイルと読んでいる。

“実”		“習”		(空白)	4	‘¥r’	‘¥n’
0x8e	0xc0	0x8f	0x4b	0x20	0x34	0x0d	0x0a

テキストデータのサンプル

一方、バイナリファイルは文字を表さないバイトが並ぶので、普通のエディタで表示させることはできないか、または表示させても意味のないデタラメなものが並んで表示される。画像データ (.gif, .png, .jpg)、音声データ (.wav, .mp3)、圧縮ファイル (.zip)、ワープロ文書データ (.doc, .rtf, .ppt)、表計算データ (.xls) などはバイナリファイルであるが、.xpm, .svg のようにテキストデータの画像ファイル形式などもある。

0xff	0xd8	0xff	0xe0	0x00	0x10	0x4a	0x46

バイナリデータのサンプル (JPEG ファイルの先頭)

プログラムソースファイルはテキストファイルであるが、コンパイルして作られる実行ファイル (.exe) はバイナリファイルである。しかし、インタプリタ言語ではコンパイルせずにソースファイルをそのままインタプリタに実行させるので、バイナリファイルは作らない。ワープロ文書のような整形文書ファイルでも、LaTeX ソースファイルや HTML ファイルはマークアップ形式のテキストファイルであり、表計算データも CSV 形式やタブ区切り形式などの形でテキストファイルとして保存できる。

バイナリ形式の利点は、

- 文字ではないデータを保存する場合はバイナリの方が小さくて済む
  (0x1234 という数値は、バイナリなら 0x12 と 0x34 に分けて表現すれば 2 バイトで済むが、テキストだと '0', 'x', '1', '2', '3', '4' の 6 バイト必要)
- 専用のソフトでしか参照できないので簡単に見られることがないし、他のソフトで使われることがない

テキスト形式の利点は、

- データを容易に参照、検索ができる
- データを処理するのに専用のソフトは必要ないので、多くのソフトで共有したり色々な処理や使い回しが可能

等であろうと思う。

バイナリデータだと処理をするのに MS–Excel などの特定のソフトの上で、そしてそれに従った操作でないと処理はできないが、テキストデータならば MS–Excel などを使わなくても AWK 等のスクリプト言語で処理できるし、しかも通常その方が作業は早く、MS–Excel が起動するのを待つ間に仕事が終わってしまうこともあるだろう。簡単なプログラミングができる人には、テキストデータを簡単なスクリプトで処理する方が楽な作業はかなりあるだろうと思う。

なお、C 言語でもテキストファイルの処理は可能であるが、多くのスクリプト言語が持つ「正規表現」や「フィールド分割」、「連想配列」などのテキストファイル処理機能がない (一から作るか、非標準のライブラリを必要とする) ので、スクリプト言語に比べると、C 言語ではテキストファイルの処理はそれほど易しいわけではない。よって、C 言語だけでなく簡単なスクリプト言語を知るのは良いだろうし、実際に多くの場面で使われている。