2016 年 07 月 29 日

計算機実習 III (2016 年度)
第 14 回: バッチファイル、AWK の応用 その 2

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

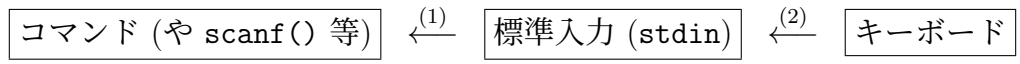
目 次

1 入力リダイレクション 1
2 パイプ 3
3 パイプ用コマンド 5
コラム: 小さなプログラムの組み合わせ 8

1 入力リダイレクション

第 4 回 (コマンドプロンプトとバッチファイルその 4) で、コマンドの標準出力の出力先をファイルにつなぎかえる出力リダイレクションを 2 種類 (>, >>) 紹介したが、リダイレクションにはもう一つ「**入力リダイレクション**」がある。

C 言語の scanf() や getchar() 等の関数は、プログラムからユーザのキーボード入力を取得する関数であるが、実際は **標準入力** (C 言語では stdin) と呼ばれる仮想デバイスを介して行われている。



出力リダイレクションと同様に、この (2) の関係を入力リダイレクションを用いて、キーボードからファイルにつなぎかえることができる (表 1)。

種類	意味
[コマンド] < [ファイル]	[コマンド] の標準入力をキーボードから [ファイル] に切り替える

表 1: 入力リダイレクション

例えば gawk は、スクリプトファイルを指定して、データファイルを指定しなかった場合、データを「標準入力」から受けとる仕様になっているので、

```
Z:¥> gawk -f script1.awk file.dat
```

は、入力リダイレクションを用いて

```
Z:¥> gawk -f script1.awk < file.dat
```

と実行しても同じことになる。

また、ひとつのコマンドの入力と出力の両方をリダイレクションでファイルにつなぎかえることもできる。その場合、

```
Z:¥> [コマンド] < [ファイル 1] > [ファイル 2]
```

か、または

```
Z:¥> [コマンド] > [ファイル 2] < [ファイル 1]
```

と書く (前者の方が自然)。こうすると、[コマンド] は [ファイル 1] の内容をキーボードからの入力のように受け取りながら実行し、その [コマンド] の画面出力が [ファイル 2] に書き出される。なお、この場合 [ファイル 1] と [ファイル 2] を同じファイル名としてしまうと予期せぬ結果を招きうる。

このように、データを標準入力からもらって、その処理結果を標準出力に書き出すプログラムを一般に **フィルタ** と呼ぶ。

例えば、sort というコマンドは、標準入力からデータを読み込み、行単位で辞書順でソートを行い、並びかえたものを標準出力に出力するフィルタなので、file1.dat が

```
2033 168 70
3001 185 85
1015 171 100
```

であるとき、

```
Z:¥> sort < file1.dat > file2.dat
```

とすると、file1.dat をソートした結果が file2.dat として保存され、file2.dat は

```
1015 171 100
2033 168 70
3001 185 85
```

となる。

なお、コマンド行の最初は [コマンド名] で始めないといけないので、上の例を

```
Z:¥> file1.dat > sort > file2.dat
```

や、

```
Z:¥> file2.dat < sort < file1.dat
```

などと実行することはできない。こうすると、file1.dat や file2.dat がそれぞれコマンド名であると解釈されてしまう (が、それらは実行できないファイルなので実際には何も実行されない)。

また、上の入出力リダイレクションを両方使った実行例と、

```
Z:¥> sort < file1.dat
Z:¥> sort > file2.dat
```

のように 2 つのコマンド行に分けたものとは全く意味が異なることに注意せよ。こちらは、sort コマンドが 2 回実行され、1 回目の sort では入力が file1.dat で出力は指定がないので、file1.dat をソートした結果が画面 (標準出力) に表示され、その出力は保存されない。2 回目の sort では出力が file2.dat で入力の指定がないので、キーボードからの入力待ちになり、Ctrl+Z を押すまでにキーボードに入力したものがソートされて file2.dat に保存されることになる。この場合、file1.dat と file2.dat は全く関係がないものになる。

上に書いたように gawk も標準入力からデータを取るフィルタとして使えるが、例えば、

```
Z:¥> gawk "{print $2,$3}" < file1.dat > file2.dat
```

は、file1.dat の 1 列目を消して、2 列目と 3 列目を file2.dat に保存するフィルタとなる。

2 パイプ

次に、複数のコマンド (特にフィルタコマンド) の連携のための「パイプ」という仕組みを紹介する。

[コマンド 1] の出力を、別の [コマンド 2] に入力させたい場合は、入出力リダイレクションと一時ファイル (tmpf.txt) を用いて

```
Z:¥> [コマンド 1] > tmpf.txt
Z:¥> [コマンド 2] < tmpf.txt
```

のようにすればできる。なお、これは p3 で説明した「2 つのコマンドに分けたもの」の例に似ているが、その違いに注意せよ。こちらは、異なるコマンドのデータを、一つのファイルで受け渡ししている。

この 2 つのコマンドのデータのやりとりを、一時ファイルを使わずに 1 行で行う仕組みが **パイプ** である (表 2)。

種類	意味
[コマンド 1] | [コマンド 2]	[コマンド 1] の標準出力を [コマンド 2] の標準入力に流し込む

表 2: パイプ

上の tmpf.txt を使う例は、パイプを使えば

```
Z:¥> [コマンド 1] | [コマンド 2]
```

と書ける。これにより、[コマンド 1], [コマンド 2] の 2 つのコマンドが並列に実行され、[コマンド 1] の標準出力への出力データが、そのまま [コマンド 2] の標準入力へと渡される。例えば、

```
Z:¥> dir | gawk "/<DIR>/"
```

とすると、dir によるカレントディレクトリ内のファイルとディレクトリの一覧の出力が、条件部分だけの一行スクリプトによる gawk への入力となり、<DIR> にマッチする行だけが出力される。結果として、カレントディレクトリにあるディレクトリの一覧が表示されることになる。

なお、標準出力に書き出さないコマンドを「|」の左側に指定しても意味はないし、標準入力を受けとらないコマンドを「|」の右側に指定しても意味はない。

パイプは多段階につなぐことも可能で、例えば

```
Z:¥> dir | gawk "/<DIR>/" | sort
```

のようにすれば、gawk の出力であるディレクトリの一覧が、さらに sort の標準入力に渡され、ソートされて出力されることになる。

パイプとリダイレクションを組み合せることも可能で、例えば、

```
Z:¥> [コマンド 1] < f.dat | [コマンド 2] | [コマンド 3] > g.dat
```

とすると、ファイル f.dat を入力とする [コマンド 1] の処理の出力が [コマンド 2] に渡され、その処理の出力が [コマンド 3] に渡され、その出力がファイル g.dat に保存されることになる。このリダイレクションとパイプの順番は入れかえることはできない。

3 パイプ用コマンド

標準的に MS–Windows に入っているパイプ用のコマンドやフィルタには、sort を含め、表 3 のようなものがある。いずれもオプションで入力データファイルを指定するが、データを標準入力から取ることができるものもある (○のついているもの)。出力は、いずれも標準出力になる。

コマンド	標準入力	指定ファイルに対する動作
type	×	ファイル (複数も可) を順に出力
more	○	入力を 1 画面ごとに表示
sort	○	辞書順にソートして出力
find "文字列"	○	ファイル (複数も可) 内の指定した文字列の含まれる行を出力
fc	×	2 つのファイルの違いを出力

表 3: フィルタやパイプ関連コマンド

これらのコマンドのオプションに関する注意:

- more, sort, find は入力ファイル名をオプションとして指定できるが、それを省略すると標準入力からデータを入力する。
- type, fc は入力ファイル名の指定を省略はできず (fc はファイルを 2 つ指定する)、標準入力からはデータを入力できない。
- sort は、/r オプションで逆順のソート、/+[n] オプションで各行の [n] の文字目以降に関するソート、となる。
- find は「find」と "文字列" の間に表 4 のオプションを書くことができる (が、ほとんど gawk の一行スクリプトで代用できるし、その方が機能は上)。

以下にいくつか例を示す。

オプション	意味
/v	指定文字列を含まない行の方を出力
/c	マッチした行数のみを出力
/n	行番号も出力
/i	大文字小文字の区別をせずに検索

表 4: find のオプション

- help コマンドによるヘルプメッセージが長い場合、「help if」のように 1 画面ずつ表示してくれるものもあるが、「help sort」のように一気に流れてしまって前の方が読めない場合がある。そのような場合、

```
Z:¥> help sort | more
```

とすると more がその画面表示を 1 画面ずつ止まりながら表示してくれる。

- 複数のファイルを more でまとめて読む場合、

```
Z:¥> more test1.txt test2.txt
```

とすると 1 ファイル毎に一旦止まる。それを

```
Z:¥> type test1.txt test2.txt | more
```

とすると、すべてのファイルを連結して一つのファイルとして読むことができる。

- 複数のデータをすべて合わせてソートしたい場合、type を利用して、

```
Z:¥> type test1.txt test2.txt | sort | more
```

のようにすることができる。

- 各行が学籍番号と成績 (整数) の 2 列のデータ file1.txt に対し、201412 で始まる行を取り出し、学籍番号でソートした結果を file2.txt に残すには、

```
Z:¥> find "201412" file1.txt | sort > file2.txt
```

のようにすればよい。find を gawk で代用して、

```
Z:¥> gawk "/^201412/" file1.txt | sort > file2.txt
```

とすることもできる。gawk の方が正規表現による細かい指定が可能だし、find がつける余計な行 (データファイル名や空行) も含まれずに済む。

sort コマンドに関する応用例を一つ紹介する。

1 節のデータ file1.dat を 2 列目の値でソートするには、このデータでは各行の 2 列目はすべて 6 文字目から始まるので、sort の /+[n] オプションを使って、

```
Z:¥> sort /+6 < file1.dat > file2.dat
```

とすればよいが、この手は 2 列目がすべての行で同じ位置から始まっていないと使えないし、sort コマンドは、辞書順のソートで数値としてのソートではない、という問題がある。実際、同じデータを 3 列目の値でソートするために

```
Z:¥> sort /+10 < file1.dat
```

とすると (3 列目は 10 文字目から始まる)、期待に反して

```
1015 171 100
2033 168 70
3001 185 85
```

と表示される。これは、辞書順では "70" や "85" という文字列よりも "100" の方が小さくなるからである。

ソートしたい列の開始位置が揃っていないデータや、辞書順ではなく数値としてソートしたい場合には、gawk を前処理と後処理に利用する方法がある。ソートしたい列の値を、辞書順でソートしても問題ない形式に変換して先頭に追加し、それを sort し、それが終わった後で追加した列を削除する、という方法である (以下は改行してあるが実際には一行で実行する):

```
Z:¥> gawk "{printf ¥"%03d %s¥n¥", $3, $0}" file1.dat | sort
    | gawk "{print substr($0, 5)}" > file2.dat
```

最初の gawk で各行の先頭に 3 列目の値を、必要なら左に 0 を追加して 3 文字の文字列にしたもの (%03d) を先頭に追加し、sort 後に 2 回目の gawk でその不要な部分を substr() で取り除いている (5 文字目以降のみ出力)。

実際、sort 後の、2 回目の gawk を行う前のデータを表示させてみると、

```
070 2033 168 70
085 3001 185 85
100 1015 171 100
```

のようになっていて、長さが違う 3 列目のデータが適切に文字列化されてソートに利用されていることがわかる。

なお、一行スクリプト内で " を使用する場合は ¥" とする必要があるし、さらにバッチファイル内で同じ一行スクリプトを書く場合は、% を %% と重ねなければいけないことに注意が必要。

コラム: 小さなプログラムの組み合わせ

今回説明したパイプやリダイレクション、そしてバッチファイルやコマンドプロンプトは、コンピュータの性能が低い時代に効率的に作業をするために考えられた仕組みであるが、単機能の小さなプログラムを組み合わせて使うことで、巨大アプリケーションに負けない複雑な処理を行うことができるようになるし、効率的だし、「コンピュータに縛られるような使い方から解放される」ような、今でも有効な考え方である。

MS–Windows では、専用ファイルを大きなソフトウェアの上で対話的に処理する、という作業が主流であるが、逆にごく簡単な処理をしたい場合でも、その大きなソフトを立ち上げないとできない、そのソフトのやり方に従った方法でしか処理ができない、自動化 (非対話型) 処理ができない、などのデメリットがある。それらはバッチファイルで小さな道具を組み合わせて作業を行う、という方法なら解消できる。必要な道具が足りない場合でも、小さな道具ならば自分で作ることは難しくないだろう。

C 言語の教科書では、外部のファイルとのデータをやりとりする場合、ファイルポインタを用いたオープン/クローズの処理や、専用の入出力関数の使用法を説明することが多いが、パイプ、リダイレクションを使うことを考えれば、ファイルポインタは不要で、標準入力 (`stdin`) からの入力、標準出力 (`stdout`) への出力を行うように書けばよい。 なお、テキスファイルを処理するような小さな道具であれば、多くの場合は C で書くよりも AWK スクリプトを書く方が易しいだろう。

MS–Windows には、パイプ処理やバッチファイル用の便利な小さな道具がやや少ないが、gawk (特に一行スクリプト) でそれらをある程度は補うこともできる。小さなプログラムが集まれば、それらの組み合わせで非常に多くの処理が可能になる。何かの作業をする際に、それを一から C 言語で書かなくても、すでにある道具を組み合わせることでできることも増えていく。卒業研究の研究室での作業なども、そういうものでできることが多い。計算機実習以外でも、バッチファイルと MS–Windows のコマンド、および gawk を組み合わせることで、自分に便利なコマンドなどを、是非色々と作ってみてもらいたい。