2016 年 07 月 22 日

計算機実習 III (2016 年度)
第 13 回: バッチファイル、AWK の応用 その 1

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

目 次

1 はじめに

残り 3 回は、まだ紹介していない gawk の便利な機能や、バッチファイルと gawk の連携などについて解説する。

そのため、コマンドプロンプトの実行例や、バッチファイル、AWK スクリプトのソース等を示す、いくつかの囲み記号の種類、使い分けについてあらためて記しておく。

- 二重枠での囲み部分 (): コマンドプロンプトでの実行例
  (通常 `Z:¥>` も左端に書かれる)
- 影付きの囲み部分 (): バッチファイルか AWK スクリプトのソースファイル
  (そのどちらであるかは内容、文脈で判断せよ)
- 一重枠での囲み部分 (): 単なる説明
  (コマンドプロンプトでも、ソースファイルの内容でもない)

2 フィールドセパレータの変更

AWK はデータファイルの各行を、空白区切りのフィールド単位 (列) で処理するが、この列の区切り文字である空白 (とタブ) は、以下のいずれかの方法で容易に変更できる。

1. gawk の実行時にオプション「-F [区切り文字]」をつける
2. gawk スクリプト (通常は BEGIN ブロック) で変数 FS に区切り文字列を代入する

-F オプションも、実際にはスクリプトの実行前に FS の値を変更する。

FS には、区切り文字となる 1 文字を指定するが、正規表現を指定することも可能である (ただし / / で囲まずに " " で囲む)。逆に FS = "" のように FS に空文字列を指定すると、データ行を「1 文字ずつ」バラバラな列データに分解する。例:

```
Z:¥> gawk -F , -f script1.awk data1.txt
```

これは、data1.txt の各行を , (カンマ) 区切りで列へ分割して処理を行う。なお、FS に代入する場合は、「FS = ","」のように区切り文字を " " で囲む必要があるが、-F オプションの後ろに書く場合は引用符は必要ない。また、この FS の値は、split() の第 3 引数のデフォルト値でもあるので、スクリプトで split() を使う場合は注意が必要。

Excel などの表計算データ (.xls 等) はバイナリデータなので、gawk で直接操作することはできないが、Excel 側でカンマ区切りのテキスト形式 (**CSV 形式** という) で保存すれば、「-F ,」を使うことで gawk での処理が可能になる。特に表計算データは列単位のデータなので gawk の処理に向いている。例:

```
BEGIN { FS = "," }
{ x += $2*$3 }
END { print x }
```

この gawk スクリプトはカンマ区切りのデータの 2 列目と 3 列目の積の値の合計を出力するが、BEGIN ブロックで FS を代入しているので、-F オプションは必要ない。

なお、CSV 形式のデータは、各列の値が " " で囲まれていることも多いので、その場合は、例えば以下のように gsub() を用いてそれを先に取り除く必要がある。

```
BEGIN { FS = "," }
{ gsub(/"/, ""); x += $2*$3 }
END { print x }
```

3 gawk 内部変数の初期値の指定

バッチファイルでは、外部からオプションを与えて、それを内部で %1, %2, ... のように利用できたが、gawk でも AWK スクリプト内部で使用する変数の初期値を、-v オプションを用いてスクリプトの外部から指定することができる (表 1)。

使用法	意味
-v [変数名]=[値]	スクリプトの実行前に [変数名] の変数に初期値として [値] を代入

表 1: -v オプション

例えば、

```
{   if (even == 1 && NR%2 == 0) print # 偶数行を出力
    if (even == 0 && NR%2 == 1) print } # 奇数行を出力
```

というスクリプトをそのまま実行すると、even の初期値は 0 だから、入力データの奇数番目の行のみが出力されることになるが、このスクリプト (test1.awk とする) を

```
Z:¥> gawk -v even=1 -f test1.awk data.txt
```

として実行すると、スクリプトの実行前に even に 1 が代入されるので、偶数番目の行が出力されるようになる。このようにすれば、スクリプトを修正しなくても、gawk スクリプトの動作を実行時に外部から切り替えられるようになる。

なお、この even の、-v で指定しない場合のデフォルト値を 0 でなく 1 にしたい場合は、スクリプトの BEGIN ブロックで以下のようにするとよい:

```
BEGIN { if (even == "") even = 1 } # デフォルト値の設定
{   if (even == 1 && NR%2 == 0) print
    if (even == 0 && NR%2 == 1) print }
```

-v での設定は BEGIN ブロックの実行前に行われるため、-v で even の値を指定した場合はその値は "" (空文字列) ではなくなるので、BEGIN ブロックの if 文による設定は行われないが、-v で値を指定しなければ even の初期値は文字列としては空文字列 ("") なので、逆に if 文により初期値が 1 に設定されることになる。

なお、BEGIN ブロックで「if (even == "")」をつけずに単に

```
BEGIN { even = 1 }
{   if (even == 1 && NR%2 == 0) print
    if (even == 0 && NR%2 == 1) print }
```

とすると、BEGIN ブロックは -v での設定の「後」に実行されるため、-v で even に何を設定してもそれを上書きして常にその初期値が 1 になってしまう。

また、変数のデフォルトの初期値は 0 なので、「if (even == "")」の代わりに「if (even == 0)」としても同じと思うかもしれないが、

```
BEGIN { if (even == 0) even = 1 }
{   if (even == 1 && NR%2 == 0) print
    if (even == 0 && NR%2 == 1) print }
```

だと、BEGIN ブロックでは、-v による設定がされていないために even が 0 なのか、-v で even を 0 に設定したのかの区別がつかないので、「-v even=0」とした場合でもスクリプト内部で even を 1 に強制的に再設定してしまうことになる。

4 AWK の内部リダイレクション

AWK の print 関数、printf 関数は通常は出力を画面に流すが、AWK 内部からその出力をリダイレクトしてファイルに落とすことができる (表 2)。複数のファイルに出

使用法	意味
print ... > "fname"	print の出力を fname という名前のファイルに出力 (上書)
print ... >> "fname"	print の出力を fname という名前のファイルに出力 (追加)

表 2: リダイレクションの形式

力を振り分けたい場合や出力ファイル名も AWK で管理したい場合は便利である。

リダイレクションの記号の使い方は、いずれもコマンドプロンプトと同じ形式だが、意味は少し違う (後述)。printf も print と同じ形式で利用できる。

例えば、各行が学籍番号 (20XXYZNNN) で始まる入力データファイルに対して、

```
{   if (/^20...1/) print > "data-m.dat"
    if (/^20...2/) print > "data-i.dat"
    if (/^20...3/) print > "data-e.dat"
    if (/^20...4/) print > "data-a.dat" }
```

というスクリプトを実行すると、学籍番号の 6 桁目 (学科) が 1 の行を data-m.dat に、6 桁目が 2 の行を data-i.dat に、という形でそれぞれのファイルに出力する。

なお、AWK スクリプトで通常ブロックを複数書いた場合は、各行に対してそれらが順に評価されるので、上のスクリプトは if を使わずに 4 つの通常ブロックを持つ形

```
/^20...1/ { print > "data-m.dat" }
/^20...2/ { print > "data-i.dat" }
/^20...3/ { print > "data-e.dat" }
/^20...4/ { print > "data-a.dat" }
```

に書くこともできる。

gawk 内部でのリダイレクション出力は、複数回 print で同じファイルに > で出力しても 1 行しか残らないわけではなく、複数回の出力が順に書き込まれ、その点はコマンドプロンプトのリダイレクションとは少し異なる。> と >> の違いは、指定したファイルが既に存在した場合に、> ではそれを「最初」に一度破棄し、>> ではそのファイルの末尾から出力し始めることのみである。

ファイルへの出力が終了したときにファイルを閉じる close 関数も用意されている。close() の引数 s は、リダイレクト先として指定したのと同じ文字列でなければいけ

関数	意味
close(s)	ファイル名 s のファイルを閉じる

表 3: 関数 close()

ない。通常は明示的に close() しなくても、スクリプトが終了すれば gawk はすべてのファイルを close() してくれるので、わざわざ close() する必要はない。

例えば、 学科、学年毎に学生情報が並んでいるデータファイルが、

- 2 列の行には「情報 3」のように学科名と学年のみが並び、
- その次の行からはその学科学年の学生データが、学籍番号、姓、名、性別、の順に 4 列で 1 行ずつ書かれている

という形式であるとき、これを学科学年毎の student-i3.txt のような名前のファイルに切り分けるには、以下のような AWK スクリプトを使えばよい。

```
BEGIN { ka["機械"] = "m"; ka["情報"] = "i"; ka["環境"] = "e";
    ka["建築"] = "a" }
NF == 2 { if (NR > 1) close(fname)
    fname = sprintf("student-%s%d.txt", ka[$1], $2); next }
{ print > fname } # $0 を fname に出力する通常ブロック
```

このスクリプトの 3 行目と 4 行目の通常ブロックは、列が 2 つのデータ行に対するもので、最後に next を実行しているためそのデータ行ではここだけが実行され、その他のデータ行は逆に 5 行目の通常ブロックのみが実行される。

4 行目では、出力ファイル名を sprintf() を使って作成して fname という変数に保存しているが、出力ファイルの名前が変わる場合それを変数に保存して使うようにすれば、データの出力部分では統一して「print > fname」とできる (5 行目)。出力ファイルを切り替える際は、現在の出力ファイルを一旦 close() して (3 行目)、ファイル名を変更すればよい (4 行目)。

なお、4 行目の「next」を消すと、2 列のデータ行にも「print > fname」を行うので、切り分けたファイルの先頭に「学科 学年」の行が書き出されることになる。

5 AWK の 1 行スクリプト

gawk の実行形式として、AWK スクリプトの内容が 1 行で書ける場合 (いわゆる **1 行スクリプト**) は、スクリプトファイルを作らなくても、以下のように実行できる。

```
Z:¥> gawk "[スクリプトの内容]" [データファイル]
```

すなわち、「-f [スクリプトファイル]」の代わりに、コマンドプロンプト上でスクリプトの内容を " " で囲んで指定する。例:

```
Z:¥> gawk "{print $2,$1}" data1.txt > data2.txt
```

は、data1.txt の 1 列目と 2 列目を入れかえたものを、出力リダイレクト (>) を利用して data2.txt に保存する (出力する)。

この実行形式は、バッチファイル内での AWK の活用の際に良く用いられる。

注意:

- 一行スクリプトの内部で文字 " を使うときは、スクリプト全体を囲む " と区別するために¥" と書く必要がある。
- 条件部分だけのスクリプトでも " " で囲まなければいけない。
- 一行スクリプトの gawk のコマンドを「バッチファイル」内に書く場合は、% を %% と書く必要がある (コマンドプロンプトで実行するだけなら % ひとつ)。

以下に 1 行スクリプトの例をいくつか紹介する (gawk コマンドとデータファイル部分を除いた 1 行スクリプト部分だけ)。

- ```
  "NR<=10"
  ```
  1 行目から 10 行目までを出力 (条件部分だけのスクリプト)
- ```
  "/^20...2/"
  ```
  指定したパターンに合う行を出力 (検索コマンド代わりに使える)
- ```
  "NF>0"
  ```
  空行を削除 (空行以外を出力することになるので)
- ```
  "{n+=$1}END{print n}"
  ```
  1 列目の数値の合計を出力 (複数ブロックもこのように 1 行に書ける)
- ```
  "{n+=NF}END{print n}"
  ```
  全体の単語数を表示
- ```
  "END{print NR}"
  ```
  全体の行数を表示
- ```
  "END{print}"
  ```
  最後の行を表示 ($0 は END ブロックで更新されないので)
- ```
  "{gsub(/。/,¥". ¥"); print}"
  ```
  全角句点をすべてピリオド+空白に変換
- ```
  "{print tolower($0)}"
  ```
  大文字アルファベットをすべて小文字に変換 (¥" に注意)
  ```

- ```
  "BEGIN{for(j=1;j<=31;j++)printf ¥"2015-07-%02d¥n¥",j}"
  ```

  今月の日付一覧を出力 (これは入力データは不要)

- ```
  "BEGIN{srand();for(j=1;j<=100;j++) print 5*rand()}"
  ```

  0.0 以上 5.0 未満の実数乱数を 100 個表示

- ```
  "BEGIN{t=systime(); while(systime()<t+10);}"
  ```

  10 秒間何もせずに待つ

- ```
  "BEGIN{print strftime(¥"%Y-%m-%d %H:%M:%S¥")}"
  ```

  現在の日付、時刻を表示

最後の 4 つの例のように、BEGIN ブロックのみのスクリプトなら自前でデータを作成するだけの作業も可能である (入力データファイルは指定しない)。

コラム: 入力デバイス

キーボードやマウスは、コンピュータを購入すると自然についてきたり、あるいはノート PC のように一体型になっているのでそれに「慣れさせられる」ことが多いと思うが、その入力デバイスが自分に合うかどうかは、作業効率や使いやすさ等に関して最も重要な問題で、本来はそこにこだわるべきものだと思う。

特にコンピュータを使用して指、ひじ、肩などに負担を感じる人は、自分に合うキーボード、マウス等を一度ちゃんと検討してみるとよい。マウスをトラックボールに変えたら、あるいはキーボードを反発の度合いや仕組みの異なるもの、配列の違うものに変えたら、ひじや肩の痛みがなくなった、入力にストレスを感じなくなった、といった話はよくあることである。

和田英一東京大学名誉教授は、以下のような有名な言葉を述べている。

> アメリカ西部のカウボーイたちは、馬が死ぬと馬はそこに残していくが、どんなに砂漠を歩こうとも、鞍は自分で担いで往く。馬は消耗品であり、鞍は自分の体に馴染んだインタフェースだからだ。
>
> いまやパソコンは消耗品であり、キーボードは大切な、生涯使えるインタフェースであることを忘れてはいけない。

(cf. http://www.pfu.fujitsu.com/hhkeyboard/dr_wada.html)