本誌正式名称は「シェルスクリプトマガジン vol.21」です

月刊 Shell Script Is The Supreme Glue Technology
FOR THE SOPHISTICATED SHELL SCRIPTERS
世界で唯一の

シェルスクリプトマガジン

よりめき版

2015 1 January Vol. 21 ¥500

TechLION 再録 ～ジョブズに騙されてる!～ 未来のライフスタイルとテクノロジー

増井俊之 × 寺薗淳也 × 瀬尾浩二郎

新連載 戸倉彩の2.5次元の世界

IT美女図鑑　近藤佑子

シェル芸に効くGNU AWK処方箋

Haskell版 Open usp Tukubai 完成させるぞ企画

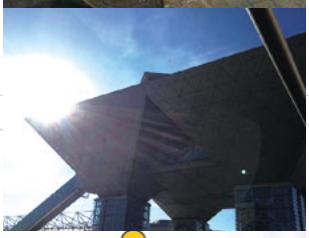
Haskellでやってはいかんのか?

新年特集

こんなシェルスクリプト、書いちゃダメだ。

Artwork created by Koji Ishizuka. © 2014 Zie's Bank .Co,. LTD All rights reserved.

CONTENTS



世界で唯一の
月刊シェルスクリプトマガジン
2015 January vol.21

04 **新年特集　こんなシェルスクリプト、書いちゃダメだ。**
後藤大地

12 **法林浩之のFITHING TALKS 第8回**
法林浩之

14 **ユニケージ開発手法　コードレビュー　第10回**
大内智明

19 **リナ女のたしなみ　第4回**
平愛美

22 **スズラボ通信　第13回**
すずきひろのぶ

28 **新連載　戸倉彩の2.5次元の世界**
戸倉彩

30 Haskell版Open usp Tukubai完成させるぞ企画
Haskellでやってはいかんのか？　第10回
上田隆一

34 組織文化を変える汗と涙の物語
アジャイル改善塾　第9回
山海一剛、喜多村貴子

36 「シェル芸に効く」
GNU AWK 処方箋　その4
斉藤博文

40 未来に活きる！現場で使える！
データモデリング　第11回
熊野憲辰

44 **IT美女図鑑-近藤佑子**

46 **Open usp Tukubaiおためし on AWS　第4回**
坪和樹

50 **人間とコンピュータの可能性　第21回**
大岩元

52 **TechLION再録　未来のライフスタイルとテクノロジー**

60 **Tech数独**

61 **新連載　姐のBENTO**

62 **吉田松蔭のことば**
シェル魔人／編集後記

本誌名称について
vol.21より「USPMAGAZINE」から「シェルスクリプトマガジン」に、変更となりました。バックナンバーは「USP MAGAZINE」としてお求めいただけます。
本誌正式名称は「シェルスクリプトマガジンvol.21」となります。

ちんじゅうちゃん

初見月

あけましておめでとうございます。新年特集はダメシェルに喝！

そのシェルスクリプト、大丈夫？

新年特集

こんなシェルスクリプト、書いちゃダメだ。

Writer：後藤大地

※本記事は、TUKUBAIコマンドを使用しております

1. ダメシェルから学べ！

シェルおよびシェルスクリプトはもっとも簡単でもっとも強力な並列プログラミング向けのソフトウェアです。開発速度の高速さ、マルチコア/メニーコアに対するスケーラビリティの高さ、動作の明快さ、システムリソースを使い切る潔さなど、どれもとってもハイレベルなソフトウェアです。

こうしてみると万能ツールのように思えますが、シェルやシェルスクリプトにははっきりとした欠点もあります（これはシェルスクリプトに限った話ではないのですが、シェルスクリプトは特にこの傾向が強いといえます）。シェルスクリプトの出来映えの良し悪しは、それを書く人のスキルに強く依存するということです。

できたものの実行速度に1,000倍以上の差があるなどザラにあります。シェルの特性を理解していない開発者が書いたコードは、遅く、非効率的で、重く、メンテナンスがしんどくて、たまったものではありません。頭がキレる方は複雑なワンライナーを書ける傾向がありますが、それは読みやすさやメンテナンス性とは真逆のものです。

大手SIerでは誰が開発しても成果物が同レベルの品質になるようにミドルウェアやフレームワークを導入し、作業を手続き化し、時間さえかければ完成するといったレベルまで機械化します。ですので、できあがるシステムは動くには動くものの、複雑で重く、特定のソフトウェアベンダに依存した、メンテナンスが困難なものになる傾向があります。

一方、シェルはそれを書く人の能力に強く依存するといった匠の世界です。中小企業はシェルのアプローチの方が導入しやすいでしょう。逆に大手SIerも今後はシェルを使う必要性に迫られるでしょう。従来の開発方式では開発に時間がかかりすぎますし、利益を出し競争力を維持するにはミドルウェアやフレームワームといった部分の価格の引き下げなどの取り組みも必要になるでしょう。従来のミドルウェアやフレームワークから脱却したいなら、シェルはそれに適したソフトウェアです。

では仮に大手SIerがシェルスクリプトベースの開発方式への移行をはかったとしましょう。この場合にもっとも大切になるのは「教育」です。少数精鋭のチームを作り、個々人のスキルをひたすらアップグレードする必要があります。

そこで今回の企画です。どういったシェルスクリプトを書けばよいのか、といった視点ではなく、どういったシェルスクリプトは「ダメなスクリプト」なのか、その視点からシェルの技術をお伝えします。スキルアップに役立てていただければと思います。

2. LINE多すぎなダメ

シェルスクリプトの仕事はコマンドを組み合わせることです。コマンドそのものがある程度の機能性を持っていますので、シェルスクリプトそのものはとてもサイズが小さくなります。特にエンタープライズシステムのように目的とする処理が明確なものであるほど、シェルスクリプトは短くなります。
目安は100行から200行くらいです。このあたりで抑えることができると、メンテナンスがしやすく、処理の流れを簡単に把握できて実に良いスクリプトです。

```
% wc -l KEIRI_SHORI_A001.SH
    3024 KEIRI_SHORI_A001.SH
%
```

プロンプト2.1 数千行に達したシェルスクリプトは抜本的に問題があるダメ

逆に、シェルスクリプトの行数が数千行に到達している場合には、いくつかの理由で設計と実装が破綻している可能性があります。その場合には次のような改善を実施すべき時にきていると言えます。

- シェルスクリプトで処理するのではなく、専用のコマンドを開発した方がよいのではないか
- 複数のシェルスクリプトに分割できるのではないか
- そもそも実装のやり方を間違っていないか

たとえば、CGIをシェルスクリプトで書いたとしましょう。次の処理をすべて単一のシェルスクリプトに書いたのであれば、それはすぐに数千行に到達するでしょう。

- クッキーおよびセッション管理
- ログイン認証
- POST/GETデータの展開処理
- 画面遷移制御
- ビジネスロジックの処理
- 入力データの健全性チェック

このケースでは、まず新しいコマンドをいくつか開発します。作成するコマンドは単機能にします。1機能1コマンドといった感じです。どういった名前でどういった単位をコマンドにしてもよいのですが、たとえば上記処理であれば次のようなコマンドを開発することになるでしょう。

- ログイン認証コマンド　login_check
- セッション管理コマンド　session_check
- 入力データチェックコマンド　data_check
- POSTデータの展開処理　get_post
- GETデータの展開処理　get_get

シェルスクリプトは画面遷移制御を実施するものがCGIのメインとなり、ビジネスロジックの処理は別途シェルスクリプト化してCGI側から呼び出すといった分割になると思います。
このように、実装すべき対象をちゃんと把握、理解し、適切な処理範囲に落とし込むことが大切です。なんでもかんでもひとつのシェルスクリプトに突っ込んでいる場合には、そもそも対象となるもの（それは業務であったり研究対象であったりさまざまだと思います）を理解していないことが考えられます。

3. 引数オーバーなダメ

次のシェルスクリプトを見て、一発で「これは嫌な感じがする」と思えるようになれば、もはやこの記事の内容は読まなくてもよいでしょう。このシェルスクリプトは実に危険な香りがします。システム運用を開始してから数年後に発動するタイプの危ないコードです。

リスト3.1 パス名展開した結果が引数の上限値を超えるダメなケース

```
#!/bin/sh

grep keyword data_2015*
```

スクリプトから推測するに、カレントディレクトリに「data_日付」といった形式のデータファイルが置かれているのでしょう。ここから特定のキーワードに一致する行を抜き出すという処理ですが、「data_2015*」がパス名展開された結果の個数が問題です。

シェルはシェルスクリプトの3行目を処理する段階で次のような処理を行います。

1. data_2015*をパス名展開。一致するすべてのファイル名またはディレクトリ名へ展開する
2. grepを実行。引数としてkeywordと展開したパス名を渡す

ここで注目すべきは、コマンドに指定できる引数には上限があるということです。無制限に引数を渡すことはできません。シェルスクリプトでは実行速度の高速化などの目的でファイルを多用しますので、data_2015*で展開される名前はシステムの運用が進むにつれてどんどん増えていくでしょう。そうなると、いつの日か、次のような事態が発生します。

```
% grep keyword data_2015*
grep: Argument list too long
%
```

プロンプト3.2 忘れた頃にやってくる大問題

だいたい問題というのは忘れた頃にやってくるものですから、こうした問題が発生するコードは仕込まないのが鉄則です。将来にわたってロバストに動作するシェルスクリプトを組み上げることが、緊急出動といったストレスフルな生活から抜け出すための秘訣です。

4. メンテナンス時発火パターンのダメ

シェルでは $$ が自身のプロセス ID に展開されます。動作している間、この値はシステムにおいてユニークな番号となりますので、この値を利用して一時ファイルを作るといったことをよく行います。典型的には /tmp/$$ といったパスでの利用です。tmp=/tmp/$$ のように変数を用意して、シェルスクリプトの内部で $tmp-out1、$tmp-out2 といったように一時ファイルとして使います。

これは便利な方法ではあるのですが、時にヒューマンエラーによるとんでもない事故を発生させる原因になることがあります。次のシェルスクリプトはとてもデンジャラスです。

リスト 4.1 メンテナンス時発火パターンのダメ

```
#!/bin/sh

tmp=/tmp/$$

ナントカ
command something > $tmp-out
カントカ

rm -rf ${tmp}*
```

ナントカカントカの部分にはいろいろと処理が書いてあるのでしょう。

次のような条件が重なると結構危険です。

- $tmp が実は使われていないことにメンテナンス中に気がついた
- 最後に rm のコードがあることに気がついていない

一時ファイルを作成せずにパイプでつないだ方がシンプルになったり高速になったりする場合など、後からコードを書き換えることもあります。この場合、$tmp の必要性がなくなり、3 行目の設定を削除する可能性があります。3 行目が削除された場合、最後の行の「rm -rf ${tmp}*」はどのように評価されるでしょうか。

$tmp は空文字列に展開されるだけですので、最後の行は「rm -rf *」となります。このシェルスクリプトを実行したプログラムのカレントディレクトリ以下が一気に削除されるわけですので、変更を実施した方やその上司の首が飛ぶか、関係者の誰かが死にかねない重大なインシデントといえます。この手のエラーでバックアップファイルが飛んだとか、システムを壊してしまったというのは実際にこれまでに例がありますので、よく気をつけておきたいところです。

5. パイプを理解していないダメ

シェルスクリプトの本質的な技術というか、これがあるからシェルスクリプトは価値があると言える機能に「パイプ」があります。けれども、実際にこれがどのように動作するものかを理解していないと、とんでもない誤ったコードを書いてしまうことがあります。たとえば次のコードを読んでみてください。seq 1 10 は echo 1 2 3 4 5 6 7 8 9 10 と同じような意味になります。while read i で変数 i に 1 から 10 までの値が順次代入され、total=$(($total + $i)) でそれを加算しているように見えます。最後に echo $total で出力していますから、標準出力には合計値である 55 が出力されると「思い込んで」しまうところです。

リスト 5.1 まったく意味がないシェルスクリプト

```
#!/bin/sh

total=0
seq 1 10 |
while read i
do
        total=$(($total + $i))
done

echo $total
```

しかし、実際にこのコードを実行して出てくる値は 0 です。これはパイプの挙動を理解していないがための誤りです。

「command1 | command2」といった使われ方をした場合、シェルはまず自分のコピーを 2 つ作成します。この段階でシェルは同時に 3 つ存在することになります。当然、それぞれ別のプロセス ID をもった個別のプロセスです。コピーされたシェルは、ひとつはその後 command1 へ、もう 1 つはその後で command2 へ変身します。この段階で初めて指定されたコマンドとして機能しようとするわけです。

上記シェルスクリプトの場合には「seq | while」ですので、まずシェルは自分のコピープロセスを 2 つ作成し、1 つはコピーされた後で seq コマンドへ化けます。もうひとつのコピーされたシェルは、そこから化けることなく while の処理をはじめます。while から done までのコードはすべてシェルで実行できる内容だからです。化ける必要はなく、そのまま処理をします。

ここで注目すべきは、while はコピーされた別のプロセスで実行されている、ということです。プロセスはお互いにメモリ空間が独立しています。別プロセスで合計値を計算したとしても、その結果はもとのシェルプロセスにはまったく影響を与えません。ですので、合計値が元のシェルに反映されることはなく、初期値として設定した 0 が出力されているわけです。

パイプは基本的に並列プログラミングを実施するものだと考えておいてください。それぞれは独立して動作しますので、パイプで連結した段階で別空間で動作するようになります。上記の例を動作するように書き換えるとすれば、次のようにファイルを利用します。

リスト 5.2 プロセスを超えるデータの受け渡しにはファイルを使う

```
#!/bin/sh

total=0
seq 1 10 |
while read i
do
        total=$(($total + $i))
        echo $total >> out
done

tail -1 out
```

ただし、このシェルスクリプト自体がそもそもシェルスクリプト的なものではないので、あんまり参考にはしないでください。処理も遅いですし、これもダメなシェルスクリプトです。

6. モニタリング機器から集計されてくるデータのサンプル

以降では具体的にエンタープライズシステムで活用されるサンプルデータを扱いながら、ダメなシェルスクリプトを見ていきましょう。

ここでは次のようなデータを扱うものとします。1 列目が機材番号（000〜999）、2 列目がモニタリングを実施した日時（yyyymmddHHDDMM）、3 列目が計測値（000〜999）です。モニタリングしているのは温度かも知れませんし、電圧や電流かもしれませんし、携帯電話通信網のルータが送ってくるデータかもしれませんし、具体的なところはなんでもよいのですが、だいたいこんな感じのデータが集計されてきます。実際にはもっと列数は多くて、数十から数百といったところです。

```
% head data
556 201501011225 001
839 201501011236 923
567 201501011228 480
651 201501011240 473
306 201501011235 765
353 201501011208 332
580 201501011258 643
140 201501011206 751
021 201501011202 602
184 201501011204 679
% gyo data
1000000
%
```

プロンプト 6.1 ナニかのモニタリングデータ 100 万件分

このデータはいわゆるトランザクションに分類されるデータの、整理をする前の状態です。機材番号ごとに整列されているわけでもなく、日時で整列されているわけでもありません。

7. シェルの利点まったく使ってなくてダメ

用意したモニタデータから、特定の機材の特定の時間帯の、計測値の平均値を求める場合を考えてみましょう。たとえば機材番号 306 に関して、2015 年 1 月 1 日 12 時台の計測値の平均値を求めてみます。よく使われている手続き型またはオブジェクト指向系のプログラミング言語（C、Java、Objective-C、C++、C#、PHP、Python、JavaScript、Perl、Visual Basic など）を使ったことがある方なら、次のようなシェルスクリプトを書くでしょう。

リスト 7.1 手続き型的な発想で書いたシェルスクリプト

```
#!/bin/sh

total=0
lines=0
while read id datetime value
do
        if [ $datetime -le "201501011259" -a \
             $datetime -ge "201501011200" ]
        then
               if [ $id = "306" ]
               then
                      total=$(echo "$total + $value" | bc)
                      lines=$(($lines + 1))
               fi
      fi
done < data
echo $(($total / $lines))
```

一見、なんら問題はないように見えます。データを 1 行ずつ読み込み、まずは日付で処理すべきデータを判定、さらに機材番号で切り分けを実施して、計測値を加算していく。最後に合計値をデータ数で割り算して平均値を出力する、というコードです。

このコードには実際に問題があるわけではないのですが、このシェルスクリプトはシェルスクリプトの利点をまったく使い切れてません。とにかく動作が遅いです。プロセッサコアが増えても性能が上がりませんし、典型的なダメシェルスクリプトと言えます。ともかく、手続き型的な書き方になっていたら、それはシェルスクリプト的ではないと思っておいてください。

8. まったくスケールしないダメ

先ほどの処理をちょっとシェルスクリプトっぽく書いてみましょう。先ほどのような処理はすべて AWK で記述できます。書くとこんな感じです。

リスト 8.1

```
#!/bin/sh

awk 'BEGIN{total=0;lines=0;}
     $2<=20150101259&&
     $2>=20150101200&&
     $1=="306"{total=total+$3;lines++;}{}
     END{printf("%d\n",total/lines)}' data
```

最初の手続き型っぽい書き方と比べるとこいつはべらぼうに速く動きますが、残念ながらマルチコア / メニーコアに対してスケールしません。1 コアでも 32 コアでも同じ性能になってしまいます。それはパイプラインを使っていないからです。複数のプロセスが並列して動作していれば、コアの数が増えれば性能も向上しますが、プロセスが 1 つしかないとスケールのしようがありません。

手続き型のプログラミング言語と比べると多少特殊ですが、AWK を使うとすべて手続き型的な書き方をすることもできます。実際、シェルスクリプトを開いたら全部 AWK だったということもあります。シェルスクリプトに慣れていないため、これまで馴染んだ書き方ができる AWK を多用してしまうわけです。しかし、それは単一のプロセスとして実行されるため、性能がスケールしません。シェルを使う場合にはシェルのやり方を覚える必要があります。

マルチコアやメニーコアで性能がスケールするように書き換えると、例えば次のようになります。

リスト 8.2 シェルスクリプトっぽく、かつ、高速に動作するシェルスクリプト

```
#!/bin/sh

tmp=/tmp/$$

grep "^306" data | self 2 3             |
awk '$1<=20150101259&&
     $1>=201501011200{print "0",$2}'    > $tmp-out

sm2 1 1 2 2 $tmp-out | self 2           > $tmp-total

echo $(( $(cat $tmp-total) / $(gyo $tmp-out) ))

rm -f $tmp-*
```

こちらは複数のコマンドをパイプで接続して高速に動作するようになっています。いくつかの高速化テクニックも取り込まれています。箇条書きするとこんな点が注目ポイントです。

- 最初に grep(1) コマンドを使ってデータの絞り込みを行っている。grep(1) はもっとも高速に動作するコマンドのひとつで、これを使って最初にデータを絞り込むのは高速動作のために定石中の定石
- self で 2 列目と 3 列目だけを選んで awk(1) へ渡している。こうすることで awk(1) の処理を高速化している
- grep、self、awk はパイプで接続され並列に動作している
- $tmp-out に書き出されたデータはすべてキャッシュに載るため、以降はオン・メモリの速度で動作している

処理にかかる時間を比べると、その差がはっきりとわかります。

表 8.3 シェルスクリプト実行速度の違い

シェルスクリプト	実行時間	高速さ
手続き型っぽい版	49.48 秒	基準
AWK のみ版	0.62 秒	約 80 倍高速
シェルスクリプトっぽい版	0.05 秒	約 1,000 倍高速

手続き型っぽい書き方をした場合と、シェルスクリプトっぽい書き方をした場合とで処理時間が 1,000 倍ほど異なっています。これは重要なことを示唆しています。たとえば手練れのシェルスクリプトエンジニアが 2 分で処理を終了するバッチ処理を書いたとします。これに対し、シェルスクリプトに慣れていないエンジニアが手続き型の要領でシェルスクリプトを書いた場合、1,000 倍遅いということは 33 時間かかるようなものを作ってしまうということになります。

夜間バッチなどで実行するとしても 33 時間では 1 日に収まっていませんから、これはまったく使えないシステムということになってしまいます。

9. メモリを使いすぎてダメ

パイプを活用してデータ処理を並列で実行するといったシェルスクリプト特有の処理に慣れてくると、なんでもかんでもパイプで接続して処理したくなります。そっちの方が高速だからです。しかし一点、ここでコマンドそのものの動作に注意が必要です。コマンドの中にはストリーム的な挙動をしつつ、実は一旦すべてデータをメモリに展開するタイプのものがあるからです。

たとえば、機材番号 306 のデータを、時刻の古いから順に整列して表示させてほしい、といった業務依頼が来たとします。この場合には次のようなコードが思いつくでしょう。

リスト 9.1 メモリ食うし遅いしダメシェルスクリプト

```
#!/bin/sh

sort -k1,2 data | grep "^306" | self 2 3
```

これはいくつかの点でダメです。まず、sort(1) コマンドを最初に実行しているところが残念です。sort(1) コマンドはその処理の特性上、すべてのデータを読み込んで一端すべてメモリに展開します。そうしないと処理を高速にこなすことができないからです。

ということは、処理するデータの量によっては sort(1) コマンドが動作できずに終了してしまうか、処理が走ったとしてもディスクとメモリとの間でデータのやり取りが無限に近い長さで実行されて処理が終わらないという事態が起こるわけです。

これをより適切なコードに書き換えると次のようになります。

リスト 9.2 これがシェルスクリプト的良コード

```
#!/bin/sh

grep "^306" data | self 2 3 | sort -k1,1
```

sort(1) コマンドの実行は一番後です。最初に可能な限り対象とするデータを絞り込む、これが鉄則です。これはメモリの使用量的な安全策である一方、実行速度にも効いてきます。この 2 つのシェルスクリプトの実行時間はこれだけ開きがあります。

表 9.3 ちょっと順番を変えるだけでこの速度差

シェルスクリプト	実行時間	高速さ
先に sort(1) している版	3.98 秒	基準
後に sort(1) している版	0.05 秒	約 80 倍高速

順番を入れ替えるだけで 80 倍の高速化が可能ということになります。このように、シェルスクリプトは書き方次第で大きく性能に差が表れます。実装面だけに絞った場合、シェルスクリプトを使ったシステム開発を成功させる秘訣は、プログラマ / コーダのレベルでこうした知識や技術を共有できるかどうかにかかっています。

10. 関数を使いすぎていてダメ

シェルスクリプトでは関数も利用できますが、関数の利用は主に 3 つの理由からあまり推奨されません（もちろんケースバイケースですが、シェルスクリプトを使った企業システムの開発という点ではあまり使いません）。次のような理由があります。

- シェルの関数は実行される文脈によって動作が変わるというクロージャ的な実装になっており、メンテナンスの段階での書き換えなどで問題を引き起こすことがある
- シェルスクリプトのように短くすぐに開発できるようなシステムでは、関数に処理を集約するということによるメリットが得にくい
- シェルスクリプトはそのシェルが実行するため処理がスケールしない。一定の機能を関数にまとめる必要がある場合、それは別のシェルスクリプトやコマンドとして開発した方がよい。その方が複数のプロセスとして実行されるため、マルチコアやメニーコアに対して性能がスケールしやすくなる

関数を利用すると、せっかく並列プログラミングが可能なシェルスクリプトの利点が生かせませんし、100 行〜200 行といったコード量ではそもそも関数の利点が得にくいと言えます。

11. 変数を使いすぎてダメ

変数を使用しないプログラミング言語というのはあまりありませんし、シェルにも変数があります。しかし、シェルではあまり変数を使い過ぎない方がよいと言えます。データは綺麗に整理してファイルで保持する。変数はその利用が必要最低限になるようにしておいた方が、何かと便利です。

12. 分岐しまくっていてダメ

シェルスクリプトでは case、if、test、&&、||、などを使って分岐処理を使えますが、これも可能な限り使わない方が良いといえますし、分岐処理の中で分岐処理を実施するといったこともぜひ避けたいところです。シェルスクリプトに落とし込む段階ではデータ処理をパイプで接続したコマンドに流し込むところまで整理しておきます。分岐処理が必要なものについては別途コマンド化するなどして、分岐処理や手続き処理が得意なプログラミング言語を使って実装しておきます。

13. 誰も読めなくてダメ

頭が切れる方は、何重にもなったネスト構造のコマンド置換や eval を活用した処理、コマンド置換や文字列処理などの組み合わせなどで一見するともはやただの記号の羅列にしか見えないコードを書くことがありますが、これもダメです。他の人が読めませんし、往々して将来の自分が理解できないという事態に遭遇することになります。

シェルスクリプトは愚直に、シンプルに、美しく、誰が読んでも理解できる形が理想です。分岐もあまりなく、上から下へ読んでいくだけで内容が理解できる、そういったレベルに維持しておくのがベストと言えます。

14. さらなる高みへ!

ここまで紹介してきたようなことがほぼほぼ頭の中にあるなら、すでにかなり高いレベルのシェルスクリプトエンジニアになっていると言えます。そこからさらにスキルアップをはかるには、ほかのシェルスクリプトエンジニアとの技術交流を持つことをオススメします。

技術は交流することで発展します。ひとりの人間が考えつく発想や実装はたかが知れたところがあります。他人のスキルや考え方、視点を得ることで、自分が持っていなかった発想方法や実装方法を得ることができますし、新しいアイディアに結びつくこともあります。

コマンドの使い方やオプションの使い方をひとつとってもそうです。便利な使い方を知っているのと知らないのとでは実装したときの簡潔さや美しさが違ってきます。ダメシェルスクリプトを乗り越えて、さらなるスキルアップを目指しましょう！

いま私たちは何を学ぶべきか

人間とコンピュータの可能性

第21回 統計処理と前提条件

written by 慶應義塾大学 名誉教授 **大岩 元**

ビッグデータのビジネス利用が始まり出した。しかし、ビッグデータを統計処理して意味のある結果を得るためには、「処理対象のデータがどの範囲のものかが特定されていて、その中からランダム・サンプリングされている」という大前提が必要である。これが無視されて、統計処理だけが精緻に実行される例が研究では非常に多い。特に人間に関わる研究において、その傾向が見られる。薬の有効性を保証する治験においても、ランダム・サンプリングが行なわれているとは言い難い。この結果、薬害問題が生じているように私は感じている。今回は、ランダム・サンプリングについて考えてみたい。

リンゴが赤いか青いかを調べる

コンピュータの普及によって、統計処理が簡単にできるようになった。従来であれば統計処理を行なうには、高度な数学を用いた統計理論を学び、面倒な計算を手作業またはプログラムを書いて行なってきた。しかし統計パッケージが普及したことにより、利用法を理解しさえすれば手軽に統計計算ができるようになった。このこと自体は、研究の発展のために喜ばしいことである。

コンピュータが自由に使えなかった60年前には、今のように体に合った既製服を買うことはできなかった。これが、多変量解析がコンピュータのお陰で実用的に使えるようになると、日本人の中からランダム・サンプリングで必要とされる人数の人体データを計測することで、日本人の体形を少数の体形に分類することが可能となり、既製服は自分の体形に合ったものを選びさえすれば、スタイルよく着心地のよいものをどこでも買えるようになった。

しかし一方で、簡単に利用できることから、統計理論の理解が不十分なまま統計処理を実行する研究が大学等で行なわれるようになってしまった。特に理論の前提となる仮定を吟味せずに、自分に都合のよい結果を得るためだけに、統計処理が行なわれている場合も見受けられる。こうした問題を考察する一例として、「リンゴは赤いか青いか」という思考実験を行なって、何が問題であるかを考えてみよう。

例題として、「リンゴは赤いか青いか」を調べる問題を考えよう。果物屋の店頭にあるリンゴから、ランダムに数十個ないし数百個のリンゴを選び、白色光を当てて反射光の波長の分布を調べれば、赤の波長と緑の波長のあたりに二つのピークが現われて、リンゴには赤いリンゴと青い（緑の）リンゴがあることが確認できる。

しかし、「リンゴは赤い」ものだと考える人が、店頭から赤いリンゴだけ集めて測定すれば、「リンゴは赤い」という結論が得られる。同様に、「リンゴは青い」と考える人が青いリンゴを集めて測定すれば、「リンゴは青い」という結論が得られる。

この二つの結論のうち、どちらか一つだけでは正しい結果を与えたとは言えない。「リンゴは赤い」と考えた人は、自分の結果を正しいと主張し、他の人は、それは間違っていると主張するであろう。自分の主張に沿った結論を出したい研究者はしばしばこのような結果を導き出しがちであるが、その結果は科学的な研究成果とは言えない。

正しい結果を出すためには、まず「リンゴ」の定義を与え、その定義にあてはまるリンゴの中からランダムに測定するリンゴを選んで、測定を行なう必要がある。その上で結論の妥当性を示す統計処理を行なうことになる。対象の定義を与えること、定義を満たすものの中からラ

ンダムに測定対象を選ぶことが、統計処理を行なうための重要な前提であり、いかに精緻な統計処理を行っても、前提が満たされなければ、科学的な結論にはなり得ない。

意味のある統計処理

人間を対象とする研究では、客観的なデータに基づいた議論を行なうのが、物を扱う場合と較べて難しい。このため、科学的研究を行なうために統計解析が利用される場合が多い。しかし、統計は大量のデータを対象とする解析方法であり、これを少数例に対して適用するためには、ランダムサンプリングが前提となる。ここで、サンプリングの対象をしっかり定義し、その中から統計処理する対象をランダムに選ぶことが必須となる。

測定対象を定義することは、客観的な手続きで行なうことではなく、研究者の主観に基づいて決定が行なわれることになる。リンゴの定義は明らかなように見えるが、対象はリンゴに限るか、それとも果物全般ではいけないのか、または果樹を対象にするのかは、何を研究するのかという研究目的と関連して決定しなければならない。科学的な研究では、研究命題（Research Question）を主観的に決めることにこそ、研究の本質がある。これが決まれば、その後の研究の手続きは客観的に行なえるが、その場合も理論の前提に常に注意を払う必要がある。

2002年に発表された教育工学の論文に「わかりやすい文章表現と文章理解」に関するものがある。この論文では、「様々な表現をした文章を提示し、その内容に関する質問を与え、正答率と解答時間によって評価した。その結果、内容表現が1つである文は、複数の内容である文より解答時間が有意に短いことを明らかにした。」ことが報告されている。結論自体は、常識的に予想されるものであって問題ない。しかし、「50字を超える文は、50字以下の文に比べ、有意に解答時間が長い」といったことを統計解析の結果として主張している。この論文の結論はどの範囲の文章に対して成り立つのであろうか。

まず、この論文には被験者がどのように選ばれたかの記述がない。おそらく著者らの大学の学生であろう。さらに問題なのは、課題文の選び方である。種々の実験条件に対して文の長さと漢字の使用率が均一になるようにしたと記述されており、4つの提示文章が例示してあるだけで，どのような文が使われているかの記述がない。これでは、実験が行なわれた文の集合に対して結論は主張できても、どの範囲でその結論が有効なのかが明らかでない。

本来であれば、例えば新聞の文章を何万文か集め、その中から課題文に適した文をランダムに選ぶというような手続きをとる必要がある。これは、現在の自然言語処理技術を用いれば、十分可能なことである。少なくとも、どのような文章に対して研究結果の主張が行なわれているのかを明らかにすることが必要であろう。

こうした内容の「研究」が、日本を代表する理工系大学で行なわれていたことが、私には信じられない。実験の手続きについては、文句がつけられないように、精密な手続きを経て行なわれているが、肝心のデータが解析に値するものでなければ、多変量解析を行ってもそれが全く意味を持たない。

発表された学会の全国大会で、この点を著者に指摘した所、私の主張を全面的に認めた。主張された内容は、実験に使用された文に対してしか成立せず、日本語文全体に成立するのではないことになると、行なわれた精緻な統計解析は全く意味を持たない。論文として全く内容が無いことになるが、査読済みで公刊された論文を取り下げることはなかったようである。また、この論文の筆頭著者の修士論文であったようだが、それについて議論が行なわれた形跡もない。

STAP細胞問題で、研究者のモラルが問われているが、10年以上前から、大学内ではあやしい研究が行なわれてきたのである。現行の学習指導要領では、統計学者の悲願である統計教育が、やっと本格的に行なわれるようになった。今までは、高校3年の最後に授業をするような学習計画になっていたため、大学受験には出題されることがなかった。このため、本格的な統計教育は大学レベルでしか行なわれなかったのである。しかし大学では、複雑な多変量解析もデータを入れれば手軽に行なえる時代が来たために、高度な統計解析ライブラリーの使い方を教えることが統計の授業の主要部になってしまったようである。コンピュータの無い時代に統計学を学んだ教授にとっては、便利な時代になったのであるが、使い方だけ習った学生は、解析の手続きだけ学ぶことになってしまう。複雑なことが簡単にできる時代は、それを使う側の教育をどうするかという大きな問題をつきつけているのである。

HANDS LAB

ユニケージ®エンジニア数 最大級!

ハンズラボはユニケージ®開発手法に特化したITソリューション企業です。東急ハンズの営業システムを刷新したノウハウを駆使し、小売業における「オーダーメイド」のシステム開発を行います。

中途採用 大募集!詳しくはHPで!

› http://www.hands-lab.com/

「ユニケージ®」は有限会社ユニバーサル・シェル・プログラミング研究所の登録商標です。

これであなたもユニケージエンジニア!

ユニケージ開発手法教育講座

「ユニケージ開発手法教育講座」は、ユニケージ開発手法におけるデータ管理の方法や、オリジナルコマンドの使用方法などをハンズオン形式で具体的に学べる講座です。UNIX の基礎からユニケージ開発手法による開発プロジェクトの進め方まで、ユニケージエンジニアとしてのトータルスキルを習得できます。

http://www.usp-lab.com/LECTURE/CGI/LECTURE.CGI

- K-BASIC ユニケージ基礎編
- K-WEB WEBアプリケーション編
- K-BATCH バッチ処理編（ウェブアプリケーション処理）
- K-ARCH ユニケージアーキテクチャ編
- K-SETUP ユニケージ開発環境セットアップ編
- K-UNYO システム運用・管理編
- K-PROJECT プロジェクトマネジメント・人材育成編
- K-SQL 速習：SQL からの移行編
- K-STAT ユニケージにおける統計コマンド編

続々と新講座も充実中!

UNIX 初心者のための講座などもご用意しています。

TechLION For Independent Engineers

技術の草原で百獣の王を目指す
エンジニアたちの新感覚トークライブ!

http://techlion.jp/

上記のサービス内容は 2014 年 12 月現在のものです。最新の情報はホームページにてご確認ください。

ISBN 9-784-904807-15-6
C3455 ￥500

9784904807156

USP 研究所
定価（本体 500円+税）

1923455005002