Shell Script ライトクックブック 2014

リッチ・ミカン 著

Rich Lab.
まつらリッチ研究所
Seeking Any Superfluous Things, since 2002

まえがき

本書制作の経緯

2005 年、本書の第一弾となる「Shell Script ライトクックブック」という同人誌を発売した。当時、UNIX サーバー管理をしながら手の空いた時に書き貯めたレシピを収録したものだが、世にある厚い本とは対照的にレシピの数は少なかった。こんな薄い本がどれだけ需要あるだろうかと不安は感じていたものの不思議とよく売れた。しかもその後この本は商業誌となり、USP 出版から「シンプルレシピ 54」として今も発売されている。こちらを買ってくれた皆さんもありがとう。

その USP 出版とは USP 研究所という会社の出版部門であるのだが、この会社が**エクストリームな変態企業（褒め言葉）**だったのである。シェルスクリプト以外の言語やデータベースミドルウェアを使わず、システム開発をやるというのだ。しかも話を聞けば、無印良品や東急ハンズ、ローソンなど、有名企業もバリバリ採用しているということで、どうやら遊びではないらしいことがわかった。

もともとこんな本を出すほどにシェルスクリプトが好きだったため USP 研究所とも付き合うようになり、シェルスクリプト総合誌 “USP MAGAZINE”（現「シェルスクリプトマガジン」）を立ち上げた。これは 2013 年まで同人誌としコミケでも頒布していたが、それを買ってくれた皆さんもありがとう。そして「日本のインターネットの父」と称される村井純や、あの “K&R” こと “The C Programming Language” の著者でもあるブライアン・カーニハンさえもビックリさせるほどの（実話）エクストリームなシェルスクリプトの世界を取材するとともに、自分自身も浸っていった。

そして 2013 年、その浸り具合は行くとこまで行ってしまった。私は **POSIX 原理主義こそ至上である**という思想を唱えるようになってしまったのである。USP 社の歴史を聞けば、昔は標準コマンドで足りない機能を sed や AWK などを駆使して作っていたのだという。別に USP 社は UNIX 哲学こそ唱えど、POSIX 原理主義を唱えているわけではない。だが今ほど UNIX 系 OS の互換性が整備されていなかった時代に、業務の都合で様々な UNIX 系 OS を渡り歩かざるを得ない必要に迫られ、どの環境でも使えるような書き方をしていたのだという。その話に感銘を受けた私が勝手に唱えているだけなのだが、実際に頑張ってみると本当にいろいろなアプリケーションが作れてしまった。2013 年は **POSIX の範囲でショッピングカートを作ることに成功**した。2014 年の今年は、ただ一つ POSIX 範囲外の curl コマンドの力を借りるだけで、東京メトロのオープンデータ（JSON）を受信し、運行情報をブラウザーに表示する Web アプリケーションも作れてしまった。

こうしてソフトウェアを作っている間に貯まったレシピを書き記したのが本書である。POSIX をしゃぶり尽くす様を見て、存分に楽しんでもらいたい。

POSIX 原理主義がすばらしい三つの理由

私は POSIX 原理主義を唱えているわけだが、単なる自己満足で唱えているわけではない。POSIX 原理主義を貫くと得られる様々な恩恵があるから唱えているのである。そのうちの最も大きな三つをここで述べよう。こ

れらの理由は、サーバー管理で日々苦労している人ほど身に滲みる内容であるはずだ。

どこでも動く

POSIX の範囲で書かれたソフトウェアは UNIX と名乗る OS 上ならどこでも動く。なぜなら、「UNIX を名乗る OS なら、最低限この仕様は守りましょう」という仕様が POSIX だからである。言いかえれば「UNIX 系 OS の最大公約数」だ。

「bash に脆弱性が見つかったので直ちに別のシェルに乗り換えたい」とか、「利用していたレンタルサーバー業者が急に倒産してしまったので別のサービスに乗り換えざるを得ないが、OS が替わってしまう」という外的要因に見舞われても痛くも痒くもない。

コンパイルせずに動く

POSIX の範囲で書かれたソフトウェアを動かすのにコンパイルという作業は不要である。なぜなら、POSIX という要件を満たすために必要なコマンドは全てコンパイルされて、揃っており、揃っているからこそ POSIX を名乗れるからだ。

それゆえに、他のソフトでありがちな、「あっちの OS に持っていったらコンパイルが通らない！」といって悩むこともないし、コンパイル作業が要らないゆえにインストール作業はコピーだけということになり、あっという間に終わる。前述のような理由でホストの引っ越しを迫られたとしても重い腰を上げる必要などない。（そもそもデータベースを使っていないので、エクスポートやインポートといった作業も無い）

10 年後も 20 年後も、たぶん動く

POSIX の範囲で書かれたソフトウェアは、10 年、20 年の規模で長期間動き続ける。なぜなら、POSIX は、全ての UNIX 系 OS に準拠させるための最低限の仕様であるから、一つ二つの OS の都合で軽々と変更するわけにはいかないからだ。

OS 独自のコマンドや他言語、データベースに依存したアプリケーションだと、利用しているそれらのソフトウェアのバージョンが 0.1 上がるだけで正常に動かなくなってしまったということが珍しくない。一方、POSIX の範囲だけで作られたアプリケーションであれば、バージョンアップの影響などまず受けない。そもそもバージョンアップ作業を強いられるのは OS くらいだ。

本書のポリシーおよび対象とする環境・ユーザー

序章の最後に、本書が対象としている読者について記しておく。

全ての UNIX 系 OS で使えること –極力 POSIX の範囲で済ませる–

本書が対象とする環境は基本的にシェルスクリプトが動く全ての UNIX 系 OS である。シェルスクリプトとは、範囲の差こそあれど UNIX 系 OS であればどこでも、しかもインストール直後から動く言語であるのだから、その全て環境のユーザーに役立つ書籍でありたい。

そのため、基本的にはどの環境でも使える手法、つまり POSIX の範囲のレシピ紹介する。より具体的には、

“IEEE Std 1003.1”[*1]で規定されているコマンドやオプション等にとどめる。Perl や Python 等、多言語などもってのほかだ。そもそも Perl を許すなら、始めから全て Perl でやってしまった方が早いだろう。

そうは言うものの、もちろん全ての UNIX 系 OS 上で本書で紹介しているレシピを試食できたわけではないため、お使いの環境よってはご賞味頂けないレシピがあるかもしれない。実のところこちらでの試食は FreeBSD 9〜10、それに CentOS 5〜7 と AIX でしか行っていない。どうかご容赦頂きたい。

シェルスクリプトの基礎が身についている方であること

本書はレシピ集である。つまり「やりたいことに対して、どのような機能を使い、また工夫をすればそれが実現できるか」を紹介する本である。従ってシェルスクリプトやコマンドにどんな機能が用意されているかを知っていなければ、本書のレシピを理解し、活用することは難しいと思う。できればその部分についてもページを割いて説明したいところではあるが、第一弾と同様に薄い本にするために省くこととした。

シェルスクリプトにまだあまり馴染みの無い方には不便を掛けてしまい大変申し訳ないが、本書のレシピを理解するのが難しいのであれば Web 上のシェススクリプトについて解説しているページなどと一緒にご覧頂きたい。

“Open usp Tukubai” と、そのクローンについて

本書を読み進めていくと “Open usp Tukubai” という用語が出てくる。これは、冒頭でも話題にした USP 研究所からリリースされているシェルスクリプト開発者向けコマンドセットの名称である[*2]。このコマンドセットは、シェルスクリプトをプログラミング言語として強化するうえで大変便利なものであり、本書で紹介するレシピのいくつかでは、そこに収録されているコマンド（Tukubai コマンド）を利用している。

しかしながらオリジナル版の Tukubai コマンドは、全てその中身が Python で書かれており、本書が提唱する POSIX 原理主義を貫くことができない。そんな中、やはり POSIX 原理主義に賛同している一人である 321516 さんから、Tukubai コマンドの何割かを POSIX で作り直したという話を教えてもらった（現在も移植中とのこと）[*3]。本書のレシピで利用している Tukubai コマンドは全て、POSIX クローン版として移植の完了しているものであるので、安心して POSIX 原理主義の実力を見てもらいたい。

おことわり

細心（最新）の注意を払ってはいるものの、間違った記憶、あるいは執筆後に仕様が変更されることによって**正しく動作しない内容が含まれている可能性**がある。不幸にもなおそのような箇所を見つけてしまった場合は下記の宛先へこっそりツッコミなどお寄せ頂きたい.

`richmikan@richlab.org`

[*1] “ieee” と “POSIX” という単語で検索すれば記載している Web ページに辿り着く。執筆時は 2013 年版が最新である。

[*2] 公式サイト→ `https://uec.usp-lab.com/TUKUBAI/CGI/TUKUBAI.CGI?POMPA=ABOUT`。尚、恐ろしいほどの処理速度を発揮する有償版 “usp Tukubai” というものも存在する。

[*3] GitHub 上で公開中→ `https://github.com/321516/Open-usp-Tukubai/tree/master/COMMANDS.SH`

目次

第 1 章

ちょっとうれしいレシピ

本章では、知っているとちょっとだけうれしいレシピを紹介する。ちなみにすごくうれしいレシピは、後ろの章で紹介するのでお楽しみに。

レシピ 1.1　ヒストリーを残さずログアウト

問題

今、`rm -rf ~/public_html/*`というコマンドで公開 web ディレクトリーの中身をごっそり消した。
こんなおっかないコマンドはヒストリーに残したくないので、今回だけはヒストリーを残さずにログアウトしたい。

おことわり

この Tips は不作法だとして異論が出るかもしれない。私個人は、何か致命的なことが起こるとは思わないものの、**ここで紹介するコマンドを打って何か不具合が起こったとしても苦情は受け付けない**ので予めご了承いただきたい。

回答

ログインしたいと思った時、次のコマンドを打てばよい。

```
$ kill -9 $$ ↵
```

解説

“`kill -9 <`プロセス *ID*`>`” とは指定プロセスを強制終了するためのコマンド書式だ。変数`$$`は、今ログインしているシェルのプロセス ID を持っている特殊な変数であるため、これを強制終了することを意味する。

強制終了とは、対象プロセスに終了の準備をさせる余地を与えず瞬殺することを意味するから、シェルに対してそれを行えば、ヒストリーファイルを更新する余地を与えずログアウトできるというわけだ。

簡単でしょ。

レシピ 1.2　sed による改行文字への置換を、綺麗に書く

問題

sed コマンドで任意の文字列（説明のため “¥n” とする）を改行コードに置換したい場合、GNU 版でない sed でも通用するように書くには

```
sed 's/¥¥n/¥
/g'
```

と書かねばならない。しかしこれは綺麗な書き方ではないので何とかしたい。

回答

シェル変数に改行コードを代入しておき、置換後の文字列をしている場所にそのシェル変数を書けば綺麗に書ける。ただし、末尾に改行コードのある文字列をシェル変数に代入するには一工夫必要だ。

まとめると、次のように書ける。

■改行コードへの置換を綺麗に書く

```
# --- sed において改行コードを意味する文字列の入ったシェル変数を作っておく ---
LF=$(printf '¥¥n_')
LF=${LF%_}

# --- 標準入力テキストデータに含まれる"¥n"を改行コードに置換する ---
sed 's/¥¥n/'"$LF"'/g'
```

解説

例えば入力テキストに含まれる “¥n” という文字列を本当の改行に置換したいという場合、sed でもちゃんとできることはできるのだが記述が少々汚くなってしまう[*1]。インデントしていない場合はまだしも、インデントしている場合の見た目の汚さは最悪だ。

```
# --- インデントしてない場合はまだマシ ---
cat textdata.txt |
sed 's/¥¥n/¥
/g'              |
wc -l

# --- インデントしている場合は汚いったらありゃしない ---
find /TARGET/DIR |
while read file; do
    cat "$file"  |
    sed 's/¥¥n/¥
/g'              |
    wc -l
done
```

これを綺麗に書く方法は「回答」で示したとおり、“¥” と改行コード（<0x0A>）の入ったシェル変数を作り、

[*1] GNU 版 sed なら独自拡張により置換後の文字列指定にも “¥n” という記述が使えるが、それは sed 全般に通用する話ではない。

それを置換後の文字列として指定してやればよい。

改行で終わる文字列の入ったシェル変数を作る

そのシェル変数を作る際、次のように即値で記述することもできる。

```
LF='¥
'
```

しかしこれでは結局、ソースコードを綺麗に書くという目的の達成はできていない。そこで、printf コマンドを使って “¥” と改行コード（<0x0A>）の文字列を動的に生成し、それをシェル変数に代入するのだが、直接代入しようとすると失敗する。それは、コマンドの実行結果を返す “$(〜)” あるいは “‘〜‘” という句が、実行結果の最後に改行があるとそれを取り除いてしまうからだ。

取り除かれないようにするには、改行コードの後ろにとりあえずそれ以外の文字の付けた文字列を生成して代入してしまう。そして、シェル変数のトリミング機能（この場合は右トリミングの “%”）を使い、先程付けていた文字列を取り除いて再代入する。この時は “$(〜)” 句を使っていないから、文字列の末尾が改行であっても問題無く代入できるのである。

シェル変数を sed に混ぜて使う場合の注意点

ここで作ったシェル変数を用いて今回の置換処理を記述する時、

```
sed 's/¥¥n/'$LF'/g'
```

と書かないように注意。$LF をダブルクォーテーションで囲まなければならない。ダブルクォーテーションで囲まれなかったシェル変数は、その中に半角スペースやタブ、改行コードがあるとそこで分割された複数の引数があるものと解釈されてしまう。つまり、“s/¥¥n/¥” と “/g” が別々の引数であると解釈され、エラーになってしまうからだ。

これは sed に限った話ではないので、コマンド引数をシェル変数と組み合わせて生成する時は常に注意すること。

レシピ 1.3　grep に対する fgrep のような素直な sed

問題

シェル変数に入っている文字列に置換されるマクロ文字列を定義し、それをテキストファイルの中に配置したい。sed コマンドを使おうと思うのだが、シェル変数にはどんな文字が入っているのかわからない。つまり sed の正規表現で使うメタ文字が入っている可能性もあるので、単純にはいかない。

回答

sed がメタ文字として解釈しうる文字を予めエスケープしてから sed に掛ける。具体的には次のコードを通すことで安全にそれができる。置換前の文字列（マクロなど）が入っているシェル変数を$fr、置換後の文字列が入っているシェル変数を$to とすると、

```
# メタ文字をエスケープ
fr=$(printf '%s' "$fr"           |
     sed 's/¥([].¥*/[]¥)/¥¥¥1/g' | # "^","$"以外の正規表現メタ文字をエスケープ
     sed 's/^¥^/¥¥^/'            | # 文字列先頭にあるメタ文字"^"をエスケープ
     sed 's/¥$$/¥¥$/'            ) # 文字列末尾にあるメタ文字"$"をエスケープ
to=$(printf '%s' "$to"        |
     sed 's/¥([¥&/]¥)/¥¥¥1/g' )    # 後方参照として意味を持つメタ文字をエスケープ

# あとは普通に sed に掛ければよい
cat template.txt | sed "s/$a/$b/g"
```

このような「素直な sed」を “fsed” という名前で GitHub に公開した[*2]ので、よかったら使ってもらいたい。まぁ、grep に対する fgrep が軽いのと違って、**この fsed は sed より軽いということは無い**のだが……。

解説

このレシピはもともと、HTML テンプレートにマクロ文字を置きたいという要望があってまとめたレシピだ。例えば、

```
<input type="text" name="string" value="###COMMENT###" />
```

という HTML テンプレート（の一部）があって、###COMMENT###の部分を、CGI 経由で受け取って今 $comment というシェル変数に入っている任意の文字列を書きたいと思った時、

```
sed "s/###COMMENT###/$comment/g"
```

と書けないのだ。なぜか？

「わかった。"をエスケープしないと HTML が不正になるからでしょ」と、気が付いたかもしれない。いや、それもそうなのだが、**むしろそのエスケープが原因で sed が誤動作**してしまう。ダブルクォーテーションを HTML 的にエスケープすると" だが、ここに含まれている&は sed の後方参照文字ではないか。

$comment の部分に、¥と&という後方参照用のメタ文字、また正規表現の仕切り文字である/が入っていると sed は誤動作する。さらに###COMMENT###の部分が、正規表現のメタ文字だったり、仕切り文字/になっていてもやはり誤動作する。これらは sed に与える前にエスケープしなければならないのだ。

正規表現のメタ文字を熟知している人なら、「回答」で示したコードを見て「あれ、(、)、{、}、+ とか、他にもいろいろメタ文字あるんじゃないの？」と思うかもしれなが、sed はこれで大丈夫。拡張正規表現モードにしない限り、他のメタ文字は手前に¥を付けることになっているからだ[*3]。

レシピ 1.4　mkfifo コマンドの活用

問題

他人のシェルスクリプトを見ていたら mkfifo というコマンドが出てきたが、これの使い方がわからないので知りたい。

[*2] https://github.com/ShellShoccar-jpn/misc-tools/blob/master/fsed

[*3] 「正規表現メモ」さんの sed に関する記述が参考になるだろう→ http://www.kt.rim.or.jp/~kbk/regex/regex.html#SED

回答

mkfifo コマンド、もとい名前付きパイプ（FIFO）は技術的にはとてもオモシロい。そこで使い方を解説しよう。

mkfifo コマンド入門

まずは同じホストでターミナルを 2 つ開いておいてもらいたい。そして最初に、片方のターミナル（ターミナル A）で次のように打ち込む。すると `hogepipe` という名のちょっと不思議なファイルが出来るので、確認してみてもらいたい。

■ターミナル A. #1

```
$ mkfifo hogepipe ↵
$ ls -l ↵
prw-rw-r--1 richmikan staff 0 May 15 00:00 hogepipe
$
```

このように `ls -l` コマンドで内容を確認してみる。行頭を見ると`-`（通常ファイル）でもない、`d`（ディレクトリー）でもない、珍しいフラグが立っている。

`p` とは一体何なのか……。そこでとりあえず cat コマンドで中身を見てみる。

■ターミナル A. #2

```
$ cat hogepipe ↵
```

すると、まるで引数無しで cat コマンドを実行したかのように（どこにも繋がっていない標準入力を読もうとしているかのように）固まってしまった。だが CTRL + C で止めるのはちょっと待ってもらいたい。ここで先程立ち上げておいたもう一つのターミナル（ターミナル B）から、今度は `hogepipe` に対して何か echo で書き込んでみてもらいたい。こんな具合に……

■ターミナル B. #1

```
echo "Hello, mkfifo." >hogepipe ↵
$
```

すると何も無かったかのように終了してしまった。今書いた文字列はどこへ行ったんだろうかと思って、ターミナル A ををを見てみると……

■ターミナル A. #3

```
$ cat hogepipe ↵
Hello, mkfifo.
$
```

先程実行していた cat コマンドがいつの間にか終了しており、ターミナル B に打ち込んだ文字列が表示されている。実はこれが mkfifo コマンドで作った不思議なファイル、「名前付きパイプ」の挙動なのだ。

つまり、

1. 名前付きパイプから読み出そうとすると、誰かがその名前付きパイプに書き込むまで待たされる。
2. 名前付きパイプへ書き込もうとすると、誰かがその名前付きパイプから読み出すまで待たされる。

という性質があるのだ。今は 1 の例を行ったが、ターミナル B の echo コマンドをターミナル A の cat コマンドより先に打ち込めば今度は echo が cat の読み出しを待つので、試してみてもらいたい。

mkfifo の応用例

こんな面白い性質がありながら、いざ用途を考えてみるとなかなか思いつかない。あえて提案するなら、例えばこういうのはどうだろうか。

- 外部 Web サーバー上に、定点カメラ映像を**プログレッシブ JPEG** 画像ファイル[*4]として配信するサーバーがある。
- ただしその Web サーバーは人気があって帯域制限が激しく、JPEG 画像を最後まで**ダウンロードするのには相当時間がかかる。**
- 上記のファイルがダウンロードでき次第、3 人のユーザーの `public_html` ディレクトリーにコピーして共有したい。でもできれば**ぼんやりした画像の段階から見せられるように**したい。

このような要求があったとして、次のようなシェルスクリプト（2 つ）を書けば解決してあげられるだろう。

■画像を読み込んでくるシェルスクリプト

```
#! /bin/sh

[ -p /tmp/hogepipe ] || mkfifo /tmp/hogepipe # 名前付きパイプを作る

¥section{30 分ごとに最新画像をダウンロードする}
while [ 1 ]; do
  curl 'http://somewhere/beautiful_sight.jpg' > /tmp/hogepipe
  sleep 1800
done
```

[*4] 最初はぼんやり表示され、データが読み込まれるどクッキリ表示される JPEG 形式である。

■名前付きパイプからデータが到来し次第、3 人のディレクトリにコピーするシェルスクリプト

```
#! /bin/sh

# 名前付きパイプからデータが到来し次第、3 人のディレクトリにコピー
while :; do
  cat /tmp/hogepipe                                          ¥
  | tee /home/user_a/public_html/img/beautiful_sight.jpg ¥
  | tee /home/user_b/public_html/img/beautiful_sight.jpg ¥
  > tee /home/user_c/public_html/img/beautiful_sight.jpg
done
```

先のシェルスクリプトは 30 分毎にループするのに対し、後のシェルスクリプトは sleep せずにループする。とはいえループの大半は、cat コマンドのところで先のシェルスクリプトがデータを送り出してくるのを待っている。

もしこの作業に名前付きパイプを使わず、テンポラリーファイルで同じことをしようとするのは大変である。なぜなら、テンポラリーファイルで行おうとする場合、後のシェルスクリプトは、画像ファイルが最後までダウンロードし終わったことを何らかの手段で確認しなければならないからだ。

使用上の注意

気をつけなければならないこともある。

1 つは、書き込む側のシェルスクリプトが

```
echo 1 >  /tmp/hogepipe
echo 2 >> /tmp/hogepipe
echo 3 >> /tmp/hogepipe
```

のように、1 つのデータを間欠的に（オープン・クローズを繰り返しながら）送ってくる場合には使えない。クローズされた段階で、読み取り側は読み取りを止めてしまうからだ。

もう 1 つは、何らかのトラブルで読み書きを終える前にプロセスが終了してしまった時の問題だ。テンポラリーファイルで受け渡しをしていたのなら途中経過が残るが、名前付きパイプだと全て失われてしまうのだ。

そういう注意点もあって、面白い仕組みではあるものの、使いどころが限られてしまうのだが。

レシピ 1.5　テキストデータの最後の行を消す

問題

あるメールシステムから取得したテキストデータがあって、その最終行には必ずピリオドがある。邪魔なので取り除きたい。
だが、「行数を数えて最後の行だけ出力しない」というのも大げさだ。簡単にできないものか。

回答

「最後の 1 行」と決まっているなら、行数を数えなくとも sed コマンド 1 個で非常に簡単にできる。下記は、メール（に見立てたテキスト）の最終行を sed コマンドで取り除く例である。

```
$ cat <<MAIL | sed '$d' ↵
やぁ皆さん、私の研究室へようこそ。      ↑
以上                                    この 3 行が元のテキスト
.                                       ↓
MAIL
やぁ皆さん、私の研究室へようこそ。
以上                                    ← 2 行目で終わっている
$
```

解説

例えば標準入力から送られてくるテキストデータの場合、普通に考えれば、一度テンポラリーファイルに書き落として行数を数えなければならないところだ。あるいは「一行先読みして……。読み込みに成功したら先読みしていた行を出力して……」とやらなければならない。どちらにしても、これを自分で実装するとなったら面倒臭い。

しかし sed コマンドは、始めからその先読みを内部的にやってくれている。だから最終行に何らかの加工を施す “$” という指示が可能である。今は最終行を出力したくないのだから、sed の中で削除を意味する “d” コマンドを使う。つまり sed で “$d” と記述すれば最終行が消えるのである。

これを応用すれば、次のようにして最後の 2 行を消すことも可能だ。

```
$ cat <<MAIL | sed '$d'| sed '$d' ↵
やぁ皆さん、私の研究室へようこそ。
以上
.
MAIL
やぁ皆さん、私の研究室へようこそ。
$
```

同様にして 3 行でも 4 行でも……。まぁ、だんだんとカッコ悪いコードになっていくが。

レシピ 1.6　改行無し終端テキストを扱う

問題

標準入力から与えられるテキストデータで、見出し行（インデント無しで大文字 1 単語だけの行と定義）を除去するフィルターを作りたい。だた、それ以外の加工をされると困る。例えば入力テキストデータの最後が改行で終わっていない場合は、出力テキストデータも最終行は改行無しのままであってもらいたい。
つまり、次のような動きをする FILTER.sh を作りたい。

```
$ printf 'PROLOGUE¥nA long time ago...¥n' | FILTER.sh ↵
A long time ago...

$ printf 'PROLOGUE¥nA long time ago...' | FILTER.sh ↵
A long time ago...$     ←元データが改行無し終端なので改行せずにプロンプトが表示されている
```

回答

`grep -v'^[A-Z]¥{1,¥}$'` というフィルターを作り、これを通せば見出し行を除去することはできる。だが、最終行が改行で終わっていなければ最後に改行コードを付けてしまうので一工夫する必要がある。

どのように一工夫すればよいか、答えはこうだ。まず目的のフィルターを通す前、入力データの最後に改行コードを 1 つ付加する。そしてフィルターを通し、その後でデータの改行コードを取ってしまえばいい。結局この問題文の目的を果たすには、具体的には次のようなコードを書けばよい。

■データの末端に余分な改行を付けないフィルター

```
printf 'PROLOGUE¥nA long time ago...' |
(cat -; echo)                          |
grep -v'^[A-Z]¥{1,¥}$'                 |
awk 'BEGIN{
       ORS="";
       OFS="";
       getline line;
       print line;
       dlm=sprintf("¥n");
       while (getline line) {
         print dlm,line;
       }
     }'
```

ただし、目的のフィルターが sed コマンドを使ったものであった場合は注意が必要。BSD 版の sed コマンドは最終行が改行終わりでなければ改行コードを付加するが、GNU 版の sed コマンドは改行コードを付加しない。この違いを吸収するため、sed コマンドだった場合には、最後の AWK コマンドの前に “`grep ^`” 等、改行を付けるコマンドを挿む必要がある。

■データの末端に余分な改行を付けないフィルター（sed の場合）

```
printf 'PROLOGUE¥nA long time ago...' |
(cat -; echo)                         |
sed '/^[A-Z]¥{1,¥}$/d'                |
grep ^                                | # この行が必要
awk 'BEGIN{
       ORS="";
       OFS="";
       getline line;
       print line;
       dlm=sprintf("¥n");
       while (getline line) {
         print dlm,line;
       }
     }'
```

解説

多くの UNIX コマンドは改行が無いと途中のコマンドが勝手に改行を付けてしまうが、純粋なフィルターとして見た場合それでは困る。どうすればこの問題を回避できるかといえば、まず先手を打って先に改行を付けてしまう。そうすると途中に通すコマンドが勝手に改行を付けることは無くなる。そして最後に末端の改行を取り除けばいいというわけだ。では、改行を末端に付けたり、末端から取り除いたりは具体的にどうやればいいのだろうか。

まず、付ける方は簡単だ。「回答」に示したコードを見れば特に説明する必要もないだろう。

一方、最後に除去をしているコードはどうやっているのか。これは AWK コマンドの性質を 1 つ利用している。AWK コマンドは、printf で改行記号¥n を付けなかったり、組み込み変数 ORS（出力レコード区切り文字）を空にしたりすれば行末に改行コードを付けずにテキストを出力できる。後ろに追加した AWK はこの性質を利用し、普段なら改行コードを出力した時点で行ループを区切るところを、行文字列を出力して改行コードを出力する手前で行ループを一区切りさせるようにしてしまう。

そうすると一番最後の行のループだけは不完全になり、最後の行の文字列の後ろに改行コードが付かないことになる。しかし、予め余分に改行を 1 個（つまり余分な 1 行）を付けておいたので、不完全になるのはその余分な 1 行ということになる。結果、元データの末端に改行が含まれていなければ末端には改行が付かないし、あれば付く。

言葉では分かり難いかもしれないが、図で解説するとこんな感じだ。

```
str_1<LF>
str_2<LF>
  :
str_n<LF 有ったり無かったり>
```

↓ (末端に改行コードを付加)

```
str_1<LF>
str_2<LF>
  :
str_n<LF 有ったり無かったり><LF>
```

↓ (加工する。各行末に必ず改行コードがあるので、勝手に付加されない)

```
STR_1<LF>
STR_2<LF>
  :
STR_n<LF 有ったり無かったり><LF>
```

↓ (各行末の改行コードが次行の行頭に移動したように扱う)

```
STR_1
<LF>STR_2
  :
<LF>STR_n<LF 有ったり無かったり>
<LF>
```

↓ (最終行の改行をトル)

```
STR_1
<LF>STR_2
  :
<LF>STR_n<LF 有ったり無かったり>
```

|| (これはつまり……)

```
STR_1<LF>
STR_2<LF>
  :
STR_n<LF 有ったり無かったり>
```

ちょっと不思議な気もするが、そういうことだ。

レシピ 1.7　IP アドレスを調べる（IPv6 も）

問題

現在自分が動いているホストの IP アドレスを全て抜き出し、ファイルに書き出したい。
ただし、知りたいのはグローバル IP アドレスだけ。

回答

一部の Linux では古いコマンド扱いされるようになった ifconfig コマンド[*5]だが、UNIX 全体の互換性を考えればまだまだ不可欠。とりあえず、下記のコードをコピペすれば大抵の環境では動くだろう。

[*5] 中には、後から追加インストールしないと存在しない Linux ディストリビューションもある。

■ifconfig から IP アドレスを抽出 (v4)

```
/sbin/ifconfig -a                                    | # ifconfig コマンドを実行
grep inet[^6]                                        | # IPv4 アドレスの行だけを抽出
sed 's/.*inet[^6][^0-9]*¥([0-9.]*¥)[^0-9]*.*/¥1/'    | # IPv4 アドレス文字列だけを抽出
grep -v '^127¥.'                                     | # lookback アドレスを除去
grep -v '^10¥.'                                      | # private(classA) を除去
grep -v '^172¥.¥(1[6-9]¥|2[0-9]¥|3[01]¥)¥.'          | # private(classB) を除去
grep -v '^192¥.168¥.'                                | # private(classC) を除去
grep -v '^169¥.254¥.'                                | # link local を除去
cat                                                  > IPaddr.txt
```

■ifconfig から IP アドレスを抽出 (v6)

```
/sbin/ifconfig -a                                          | # ifconfig 実行
grep inet6                                                 | # IPv6 行抽出
sed 's/.*[[:blank:]]¥([0-9A-Fa-f:]*:[0-9A-Fa-f:]*¥).*/¥1/' | # IPv6 抽出
grep -v  '^::1$'                                           | # loopback 除去
grep -v  '^¥(0¥+:¥)¥{7¥}0*1$'                              | # loopback 除去
grep -vi '^fd00:'                                          | # private 除去
grep -vi '^fe80:'                                          | # link local 除去
cat                                                        > IPaddr.txt
```

解説

ifconfig の出力を、ループ文や if 文などを使って 1 つ 1 つパースするようなコードを書くと長く複雑になりがち。しかしパイプと複数のコマンドを駆使すればご覧のとおり、短くてわかりやすくなる。**パイプを使えば「スモール・イズ・ビューティフル」**というわけだ！

シェル変数で受け取りたい場合は？

上記のコードはファイルに出力する場合だったが、シェル変数で受け取りたいこともあるだろう。その場合の方法は 2 つある。ただ、取得できた IP アドレスが v4、v6 それぞれ複数ある場合でも 1 つの変数に入るので後で適宜分離すること。

(1) 全体を$(〜) で囲む

方法その 1 は、全体を$(〜) で囲み、サブシェル化してしまうというものだ。

■グローバル IPv4 アドレスを取得後、変数に代入

```
ipv4addrs=$(/sbin/ifconfig -a                              |
            grep inet[^6]                                  |
            sed 's/.*inet[^6][^0-9]*¥([0-9.]*¥)[^0-9]*.*/¥1/' |
            grep -v '^127¥.'                               |
            grep -v '^10¥.'                                |
            grep -v '^172¥.¥(1[6-9]¥|2[0-9]¥|3[01]¥)¥.'    |
            grep -v '^192¥.168¥.'                          |
            grep -v '^169¥.254¥.'                          )
```

パイプ（“|”）で繋がっている一連のコマンドを囲めばよい。IPv6 の場合も同様だ。

(2) シェル関数にしてしまう

あちこちで使い回したい場合はシェル関数化するのがよいだろうか。シェル関数化したら同じくそれを$(〜)で囲めばシェル変数に代入もできる。

シェル関数化して、あたかも外部コマンドであるかのように用いる例を示す。

■グローバル IPv4 アドレス取得のためのシェル関数

```
get_ipv4addrs() {
  /sbin/ifconfig -a                                    |
  grep inet[^6]                                        |
  sed 's/.*inet[^6][^0-9]*¥([0-9.]*¥)[^0-9]*.*/¥1/'    |
  grep -v '^127¥.'                                     |
  grep -v '^10¥.'                                      |
  grep -v '^172¥.¥(1[6-9]¥|2[0-9]¥|3[01]¥)¥.'          |
  grep -v '^192¥.168¥.'                                |
  grep -v '^169¥.254¥.'
}

num_ipv4=$(get_ipv4addrs | wc -l)
echo "現在持っているグローバル IPv4 アドレスの数:" $num_ipv4
```

補足

このレシピで紹介したコードの ifconfig コマンドは/sbin にあること前提で絶対パス指定している。これは Linux で使う場合の対策である。

多くの Linux ディストリビューションでは、一般ユーザーに/sbin へのパスを設定していない。そのため大抵/sbin の中に置かれている ifconfig コマンドが見つからないのだ。もし、/sbin には無いかもしれない環境も考慮するのであれば、環境変数 PATH に、/sbin、/usr/sbin、/etc[*6]あたりを追加しておくとよいだろう。

レシピ 1.8　YYYYMMDDhhmmss の各単位を簡単に分離する

問題

20140919190454 → 2014 年 9 月 19 日　19 時 4 分 54 秒
というように、年月日時分秒の 14 桁数字を任意のフォーマットに変換したいが、AWK で substr() 関数を 6 回も呼ぶことになり、長ったらしくなるし、面倒くさい！簡単に書けないのか。

回答

まず正規表現で数字 2 桁ずつに半角スペースで分離し、先頭の 2 組（4 桁）だけ結合し直す。すると年月日時分秒の各要素がスペース区切りになるので、AWK で各フィールドを取り出せば如何様にでもフォーマットできる。

次のコードを実行すれば、2014 年 9 月 19 日　19 時 4 分 54 秒という文字列に簡単に変換できる。

[*6] AIX など、/etc に置いてある OS なんてのがあるのだ。

```
echo 20140919190454                                                    |
sed 's/[0-9][0-9]/ &/g'                                                |
sed 's/ ¥([0-9][0-9]¥) /¥1/'                                           |
awk '{printf("%d 年 %d 月 %d 日 %d 時 %d 分 %d 秒¥n",$1,$2,$3,$4,$5,$6);}'
```

AWK コマンド 1 つで行うことも可能だ。

```
echo 20140919190454           |
awk '
  gsub(/[0-9][0-9]/, "& ");
  sub(/ /, "");
  split($0, t);
  printf("%d 年 %d 月 %d 日 %d 時 %d 分 %d 秒¥n",t[1],t[2],t[3],t[4],t[5],t[6]);
}'
```

解説

ちょっと頭を捻ってみよう。アイデアとしては、正規表現で 2 桁ずつの数字にスペース区切りで分解した後、最初の 4 個は戻してやればいいわけだ。つまり正規表現フィルターに 2 回掛けるわけだが、1 つ目はグローバルマッチで、2 つ目は 1 回だけマッチさせるようにすると、西暦だけ都合よく 4 桁にできる。

これで年月日時分秒が各々スペース区切りになっているので、AWK で受け取れば自動的に$1〜$6 に格納されるし、あるいは 1 個の AWK の中で加工していたのであれば split() 関数を使って配列変数に入る。

このテクニックを知らないうちは、

```
echo 20140919190454 |
awk '
  Y = substr($0, 1,4);
  M = substr($0, 5,2);
  D = substr($0, 7,2);
  h = substr($0, 9,2);
  m = substr($0,11,2);
  s = substr($0,13,2);
  printf("%d 年 %d 月 %d 日 %d 時 %d 分 %d 秒¥n",Y,M,D,h,m,s);
'
```

と書かざるを得なかったのだから、カッコいいコードになったと思う。

レシピ 1.9　祝日を取得する

問題

平日と土日祝日でログを分けたい。土日は計算で求められても、祝日は、春分の日や秋分の日など計算のしようがないものもある。どうすればいいか？

回答

祝日を教えてくれる WebAPI に問い合わせて教えてもらう。

Google カレンダーを使うのが便利だろうということで、Google カレンダーから祝日を取得するシェルスクリプトの例を示す。

■get_holidays.sh

```
#! /bin/sh

# この URL は
# Google カレンダーの「カレンダー設定」→「日本の祝日」→「ICAL」から取得可能（2014/11/29 現在）
url='https://www.google.com/calendar/ical/ja.japanese%23holiday%40group.v.calendar.google.com/public/basic.ics'

curl -s "$url"                              |
sed -n '/^BEGIN:VEVENT/,/^END:VEVENT/p'     |
awk '/^BEGIN:VEVENT/{                       # === iCalendar(RFC 5545) 形式から日付と名称だけ抽出 ===
       rec++;                               #
     }                                      #
     match($0,/^DTSTART.*DATE:/){           # DTSTART 行は日付であるから
       print rec,1,substr($0,RLENGTH+1);    # 「レコード番号 "1" 日付」にする
     }                                      #
     match($0,/^SUMMARY:/){                 # SUMMARY 行は名称であるから
       s=substr($0,RLENGTH+1);              # 「レコード番号 "2" 名称」にする
       gsub(/ /,"_",s);                     #
       print rec,2,s;                       #
     }'                                     |
sort -k1n,1 -k2n,2                          | # レコード番号＞列種別 にソート
awk '$2==1{printf("%d ",$3);}               # # 1 レコード 1 行にする
     $2==2{print $3;       }                #
     '                                      |
sort                                        # 日付順にソートして出力
```

実行例

試しに、平成 26 年度の祝日一覧を求めてみる。

■実行結果

```
$ get_holidays.sh ↵
20130101 元日
    :
（途中省略）
    :
20141123 勤労感謝の日
20141124 勤労感謝の日_振替休日
20141223 天皇誕生日
    :
（途中省略）
    :
20151223 天皇誕生日
$
```

Google カレンダーは、当年とその前後 1 年の祝日一覧を教えてくれる。ご覧のとおり、振替休日が発生する場合は元の日付に加えて振替日も示してくれる。日付だけが欲しくて名称が邪魔な場合は最後の sort コマンド

の後に、| awk {print $1}などを付け足せばよいので簡単だ。

解説

問題文にもあるが、日本の祝日は、その全てを計算で求めることができない[*7]。春分の日、秋分の日の二祝日は毎年の天文観測によって二年後の月日を決めるように定められているからだ。計算で求められないことが明らかなら、知っている人に聞くしかない。そこで WebAPI を叩くというわけだ。

いくつかのサイトが提供してくれているが、Google カレンダーを使うのが手軽だろう。

iCalendar 形式

祝日情報をどういう形式で教えてくれるかというと、**iCalendar(RFC 5545) 形式**である。Google カレンダー自体は独自の XML 形式や HTML 形式でも教えてくれるのだが、iCalendar 形式はシンプルだし、きちんと規格化されているので、情報源を他サイトに切り替える可能性を考慮するならこの形式を選択しておくべきである。

iCalendar 形式の詳細についてはもちろん RFC 5545 のドキュメントを見れば載っているし、日本語解説は野村氏が公開している「iCalendar 仕様」[*8]が参考になる。ただ、例示したソースコードの説明に必要な項目だけはここに記しておこう。

まず、この形式は HTML タグと同様にセクションの階層構造になっている。ただし、HTML のようなタグのインデントは許されず、タグ（HTML 同様、こう呼ぶことにする）名は必ず行頭に来る。

今回注目すべきセクションは “VEVENT” でありここに祝日情報が入っているため、まずこのセクションの始まり（BEGIN:VEVENT）と終わり（END:VEVENT）でフィルタリングする。今回は祝日の日付と名称が欲しいだけなので、それらが収められているタグ（“DTSTART” と “SUMMARY”）行だけになるよう、さらにフィルタリングする。あとは、VEVENT セクションの中にあるこれら日付と名称の値を取り出して、1 つ 1 つの VEVENT 毎に横に並べれば目的のデータが得られる。

Google カレンダーの URL

ソースコードの中にもメモしているが、祝日一覧を返してくる WebAPI の URL は 2014/12/20 現在、次のように辿れば見つけることができる。Google の都合によって将来移動する可能性もあるので、参考にしておいてもらいたい。

1) ログインして Google カレンダーを開く
   （ただし最終的に得られた URL 自体はログインせず利用可能）
2) （歯車マークアイコンの中の）「設定」メニュー
3) 画面上部に「全般」「カレンダー」とある「カレンダー」タブ
4) 「日本の祝日」リンク
5) 「カレンダーのアドレス」行にある”ICAL”アイコン

[*7] 天文学データを入れれば「予測」することは可能だが、サーバー管理者にとって現実的な話ではない。

[*8] http://www.asahi-net.or.jp/~ci5m-nmr/iCal/ref.html

参照

→ RFC 5545 文書[*9]

レシピ 1.10　ブラックリストの 100 件を 1 万件の名簿から除去する

問題

今、約 1 万人の会員名簿（members.txt）と、諸般の事情によりブラックリスト入りしてしまった約 100 人会員名一覧（blacklist.txt）がある。会員名簿からブラックリストに登録されている会員のレコードを全て除去した、「キレイな会員名簿」を作るにはどうすればいいか。
ただし、各々の列構成は次のようになっている。

members.txt　**1** 列目:会員 ID、**2** 列目:会員名
blacklist.txt　**1** 列目:会員名（1 列のみのデータ）

回答

SQL の JOIN 文と同様に考え、それに相当する UNIX コマンドの “join” を活用する。この場合、「会員名」で外部結合（OUTER JOIN）し、結合できなかった行だけ残せばよい。

■ブラックリスト会員を除去するシェルスクリプト（del_blmembers.sh）

```
#! /bin/sh

# 結合に使う列は予めソートしておかなければならない（ここでは「会員名」）
sort -k1,1 blacklist.txt   > blacklist1.tmp

cat members.txt                                                 |
sort -k2,2                                                      | #・「会員名」でソート
join -1 1 -2 2 -a 2 -e '*' -o 1.1,2.1,2.2 blacklist1.tmp -      | #・B.L. を右外部結合
awk '$1=="*"{print $2,$3;}'                                       #・null 相当値のある
                                                                  #   行だけ抽出
rm blacklist1.tmp
```

解説

もし join コマンドを知らなかったらどうプログラミングするだろう。恐らくブラックリスト会員を while 文でループを回し、全会員のテキストから 1 行ずつスキャン（grep -v）してくということをするのではないだろうか。だが、それはあまりにも効率が悪すぎる。

あまり知られていないかもしれないが UNIX には join コマンドというものがあり、リレーショナルデータベースと同様の作業ができるのだ。リレーショナルデータベースを使い、SQL 文でこの作業をやってよいと言われれば、外部結合（OUTER JOIN）を用いるという発想はすぐに出てくると思う。

ここから先は join コマンドのチュートリアルを行いながら解説を進めていくことにする。

[*9] https://tools.ietf.org/html/rfc5545

join コマンドチュートリアル

実際にデータを作って JOIN することで、join コマンドの使い方を見ていこう。

まずはデータを作る

さすがに 1 万人分のサンプルネームを生成するのは大変だ。そこで/dev/urandom を用いた 4 桁の 16 進数を便宜上の名前ということにして、そういう会員名簿を作ってみる。こんなふうにしてワンライナーでササっと作ろう。

■ダミーの会員リスト（members.txt）を作る

(註) 第 1 列に会員 ID、第 2 列に (便宜上の)「名前」が入ったデータを作る

```
$ dd if=/dev/urandom bs=1 count=20000 2>dev/null | ↵
> od -A n -t x2 -v                               | ↵
> tr ' ' '¥n'                                    | ↵
> grep -v '^$'                                   | ↵
> awk '{printf("ID%05d %s¥n",NR,$0)}'             > members.txt
$
```

■ダミーのブラック会員リスト（blacklist.txt）を作る

(註) ブラックリストの (便宜上の)「名前」が入ったデータを作る

```
$ dd if=/dev/urandom bs=1 count=200 2>/dev/null | ↵
> od -A n -t x2 -v                              | ↵
> tr ' ' '¥n'                                   | ↵
> grep -v '^$'                                  > blacklist.txt
```

できたら、データの中に 16 進数 4 桁があることを確認しておこう。これがダミーの名前である。ただし前者（members.txt）は、次のように名前の手前に会員 ID が振られているはずだ。

```
ID00001 9fc6
ID00002 e13d
ID00003 6575
ID00004 1594
ID00005 1629
      :
```

ブラック会員除去をする

これらのファイルを冒頭のシェルスクリプト（del_blmembers.sh）に掛ければよい。結果をファイルに保存して、元のファイルと行数を比較してみよう。

```
$ ./del_blmembers.sh > cleanmembers.txt ↵
$ wc -l cleanmembers.txt ↵
    9988 cleanmembers.txt
$ wc -l members.txt ↵
   10000 members.txt
$
```

この例では、10000 人の会員のうち 12 人がブラックリストに登録されていたことがわかった。

join コマンド解説

チュートリアルも済んだところで、join コマンドの解説に移る。

元のコードの join の意味を説明していこう。

```
join -1 1 -2 2 -a 2 -e '*' -o 1.1,2.1,2.2 blacklist1.tmp -
```

まず、最後の 2 つの引数を見てもらいたい。これは結合しようとしている 2 つのテキストデータを指示している。2 つのテキストのうち左に記したもの（この例では blacklist1.tmp）が左から、右に記したもの（この例では標準入力）が右から結合されることになるが、それぞれ「1 番」、「2 番」という表番号が与えられることを頭に入れておいてもらいたい。

さて、以降はオプションを先頭から順番に説明していく。

`-1`、`-2` というオプションは、JOIN しようとする 2 つの表のそれぞれ何番目の列を見るかを指定するものだ。1 番の表は 1 列目を、2 番の表は 2 列目を見て、それらが等しい行同士を JOIN せよという意味である。

`-a` というオプションは、外部結合のためのものであり、JOIN できなかった行についても出力する場合はその表番号を指定する。`-a 1` とすれば左外部結合（LEFT OUTER JOIN）を意味し、`-a 2` とすれば右外部結合（RIGHT OUTER JOIN）を意味する。もし、完全外部結合（FULL OUTER JOIN）にしたければ`-a 1 -a 2` と`-a` オプションを 2 度記述する。

`-e` というオプションも外部結合のためのものである。JOIN できずに NULL になった列に詰める文字列を指定する。テキスト表記の場合は NULL を表現できない[*10]ので、このオプションによって NULL 相当の文字列を定義する。

`-o` というオプションは、出力する列の並びを指定するためのものである。SQL では SELECT 句の直後で出力する列の並びを指定するが、あれと同じものだ。何番の表の何列目を出力するのかをカンマ区切りで列挙していく。

本当はこの他に、`-v` オプションというものがあり、これが指定された場合は **JOIN できなかった行だけ表示**するようになる。例えば`-v 1` と書けば JOIN できなかった 1 番の表の行だけが表示される。勘のいい読者なら気づくと思うが、例示したシェルスクリプトは実は次のようにもっと簡単に書けるのだ。

[*10] 厳密にできないわけではない。`-e` オプションを指定しなかった場合は何も詰めるものが無いので半角スペースが連続した箇所ができる。だがそれはわかりにくい。

```
join -1 1 -2 2 -a 2 -e '*' -o 1.1,2.1,2.2 blacklist1.tmp - |
awk '$1=="*"{print $2,$3;}'

   ↓

join -1 1 -2 2 -v 2 blacklist1.tmp - |
```

ちなみに、もし SQL の SELECT 文で同じことをするなら、次のように書ける。

```
SELECT
  MEM."会員 ID",
  MEM."会員名"
FROM
  blacklist AS MEM
    RIGHT OUTER JOIN
  members   AS BL
    ON BL."会員名" = MEM."会員名"
WHERE
  BL."会員名" IS NOT NULL
ORDER BY
  MEM."会員 ID" ASC;
```

このようにして SELECT 文でできることは、join を始め、sed、AWK、grep などを使えば UNIX コマンドでも大抵できる。ついでに言うと、SELECT 文でデータの流れを追えば、FROM 句→ (RIGHT OUTER JOIN 句) → WHERE 句→ ORDER BY 句→ (最初に戻って)SELECT の直後、であるが、シェルスクリプトの場合はほぼ上から下へ一直線であるのが個人的には好きだ。

join コマンド使用上の注意

利用する場合は一つ注意しなければならない点がある。日本語ロケールになっている場合は、**デフォルトで全角空白も列区切り文字として解釈**してしまう。例えば人名フィールドがあって姓名が全角スペースで区切られている場合には注意が必要だ。

そのような場合は、`export LC_ALL=C` などとして C ロケールにしておくか、`-t` オプションを使って区切り文字をしっかり定義しておかくこと。

参照

→レシピ 2.4（sort コマンドの基本と応用とワナ）
→レシピ 5.6（ロケール）

第 2 章

利用者の陰に潜む、様々な落とし穴

シェルスクリプトや UNIX コマンドは「クセが強い」とよく言われる。それも一種の個性であるといえばそれはそれでアリなのだが、その個性を知らぬまま使うと思わぬ落とし穴にはまってしまう。

本章では、UNIX 入門者・中級者がはまりがちな、各種コマンドや文法の落とし穴を紹介していく。

レシピ 2.1　【緊急】false コマンドの深刻な不具合

問題

false コマンドに深刻なセキュリティーホールがあると聞いたが、一体どういう事か？

回答

2014 年の今年、コンピューターセキュリティー史上一、二を争うであろうとても深刻なセキュリティーホールが見つかった。それがなんと false コマンドであった。後述の情報を読み、直ちに対応してもらいたい。

詳しい経緯

2014 年 4 月 1 日、“Single UNIX Specification”[*1]を策定している The Open Group[*2]が、false コマンドに不具合を見つけたことを発表した。しかもそれは、コンピューターセキュリティー史上一、二を争うほどに深刻で、これまで一生懸命対策を講じてきた世界中のセキュリティー対策者達を一気に脱力させるほどのインパクトだという。

その理由の一つはまず、影響範囲があまりにも広いということ。なんと **false コマンドを実装しているほぼ全ての UNIX 系 OS** がこの問題を抱えており、与える影響は計り知れないというのだ。

さらに深刻な理由は、この問題が発生したのは恐らく false コマンドの最初のバージョンで既にあったということである。最初の POSIX には既に false コマンドが規定されており、それは 1990 年のことであるから、**どう短く見積もっても 24 年間この問題が存在していた**ことになる。

false コマンドは、実行すると「偽」を意味する戻り値を返すだけというこれ以上無いほどに単純なコマンドであったために、まさかそこに脆弱性があろうとは長年誰も気付かなかったのである。

[*1] http://www.unix.org/what_is_unix/single_unix_specification.html

[*2] http://www.opengroup.org/

不具合の内容

肝心の不具合の内容であるが、それは次のとおりだ。発表内容を引用する。

> 現在の false コマンドの man によれば、このコマンドは戻り値 1(偽) を返すとされています。
>
> しかしそれは、当然 1 を返してくるものと期待しているユーザーに対して**正直な動作**をしており、これは false(偽る) というコマンド名に対して実は不適切な動作をしています。

つまり、false というコマンドの名に反してユーザーの命令に忠実に動いてしまっていたのだ。これは、「名は体を表す」が重要とされるコマンド名として、ましてやそれが POSIX で規定されるコマンドとして、あってはならないことだ。

今後の対応方針

今回の不具合報告に合わせ、The Open Group のスポークスマンは次の声明を発表した。

> 確かに前述のとおりの不具合は見つかったものの、false というコマンドの長い歴史からすればとても「いまさら」な話である。また、あまりに普及率が高いコマンドで及ぼす影響も甚大であることからも、もし修正を実施したとなれば「いまさら仕様を変えるなよ」と世界中から白い目で見られることは必至である。
>
> 我々もいまさらそんな苦労と顰蹙を買いたくないし、それに、今日 (4/1) が終われば世の中もきっと我々の発表を無かったことにしてくれるに違いないと思うので、とりあえず今日をひっそり生きようと思う。

このように The Open Group は**「それは断じて仕様である」メソッドの発動を示唆**しており、false コマンドの動作は結局そのままになるものと見られている。従って、この問題に対しては各ユーザーが個別に対応し続けて行かねばならぬようだ。

情報元;-p → `https://twitter.com/uspmag/status/450658253039366144`

レシピ 2.2　名前付きパイプからリダイレクトする時のワナ

問題

次のコードを実行したものの、やっぱり cat は取り消そうと思って `killall cat` を実行したら失敗した。おかしいなと思って ps コマンドで cat のプロセスを探しても見つからなかった。どこへ行ってしまったのか？

```
$ mkfifo "HOGEPIPE" ↵
$ { cat <"HOGEPIPE" >/dev/null; } & ↵
```

回答

cat を起動する子シェルの段階で処理が止まっており、cat コマンドはまだ起動していない。もしその子シェルを kill したいのであれば次のようにして、jobs コマンドでジョブ ID を調べ、そのジョブ番号を kill する。

```
$ mkfifo "HOGEPIPE" ↵
$ { cat <"HOGEPIPE" >/dev/null; } & ↵
$ jobs ↵
[1] + Running                 cat <HOGEPIPE >/dev/null
$ kill %1 ↵
$ ↵
[1]   Terminated              cat <HOGEPIPE >/dev/null
$
```

あるいは、jobs コマンドに-l オプションを付けて子シェルのプロセス ID を調べ、そのプロセス ID を kill してもよい。

```
$ mkfifo "HOGEPIPE" ↵
$ { cat <"HOGEPIPE" >/dev/null; } & ↵
$ jobs -l ↵
[1] + 13742 Running           cat <HOGEPIPE >/dev/null
$ kill 13742 ↵
$ ↵
[1]   Terminated              cat <HOGEPIPE >/dev/null
$
```

解説

なぜ cat コマンドはまだ起動していなかったのか。これは、シェルの仕組みを知れば理解できるだろう。

シェルは、コマンドを実行する際、いきなりコマンドのプロセスを起動はしない。まず自分の分身である「子シェル」を生成する。そして、exec システムコールによって、それを目的のコマンドに変身させるという手順を取るのだ。

なぜそのようにしているかといえば、コマンドを呼ぶ前にシェル側で準備作業が必要だからだ。その一つがリダイレクションである。コマンドの前後に記された “<”、“>”、“>>” などといった記号で読み込みまたは書き込みモードでのファイルオープンが指定されたらその作業はシェルが受け持つことになっている。シェルはリダイレクション記号で指定されたファイルを標準入出力に接続し、それができてからコマンドに変身しようとする[*3]。ちなみにリダイレクションが指定されなかった場合は、特にファイルに接続することはないが、標準入出力および標準エラー出力をオープンするという作業はデフォルトで行っている。

さて今回の場合、オープン対象は “HOGEPIPE” という名前付きファイルであるがこれはレシピ 1.4（mkfifo コマンドの活用）で説明したとおり、データが書き込まれないうちにオープンしようとしたり読み込もうとすると、データが来るまで延々と待たされることになってしまう。それを子シェルがやっているものだから、cat に変身することができず、cat プロセスが未だに存在しないというわけだ。

[*3] もし cat コマンドの後に別コマンドが “|” や “&&” や “;” 等でさらに書かれていた場合は、返信せずに更に孫シェルを起動してそれらを順番に実行しようとする。

実際にデータを流してみると

では、パイプにデータを流し始めてみたらどうなるか見てみよう。“{ cat <"HOGEPIPE" >/dev/null; } &” まで実行したら、今度は kill せずに yes コマンドあたりを使ってデータを流し込み続けてみてもらいたい。

```
$ mkfifo "HOGEPIPE" ↵
$ { cat <"HOGEPIPE" >/dev/null; } & ↵
$ yes >"HOGEPIPE" & ↵
$
```

そして、ps コマンドで関連プロセスを確認してみる。

```
$ ps -Ao pid,ppid,comm | egrep $$'|'cat | egrep -v grep'|'ps ↵
$ 13510 12883 sh
$ 13839 13510 cat
$ 13840 13510 yes
$ kill 13840 ↵
$
```

左から自プロセス ID、親プロセス ID、自プロセスのコマンド名を表示しているが、今度は cat コマンドが存在していることがわかる。尚、**yes コマンドからデータを流し続けていると無駄な負荷がかかるので。確認したら速やかに yes コマンドを kill すること。**

リダイレクションでない場合は cat の起動まで進む

先程はリダイレクションを用いたために、子シェルでつかえていた。では、cat コマンドの引数として名前付きパイプを指定したらどうなるだろうか。

```
$ mkfifo "HOGEPIPE" ↵
$ { cat "HOGEPIPE" >/dev/null; } & ↵
$ killall cat ↵
$ ↵
[1]   Terminated              cat <HOGEPIPE >/dev/null
$
```

今度は killall が成功した。名前付きパイプ “HOGEPIPE” を開くのが cat コマンド自身の仕事になったからだ。先程の説明を踏まえれば理解は容易だろう。

参照

→レシピ 1.4（mkfifo コマンドの活用）

レシピ 2.3　全角文字に対する正規表現の扱い

問題

正規表現を使って全ての半角英数字（[[:alnum:]]）を置換しようとしたら、全角英数字まで置換されてしまった！全角文字はそのままにしたいのだが、どうすればいいのか？

回答

それはロケール系環境変数が日本語の設定になっているためである。従って半角英数字だけを置換対象にしたいのであれば、それらの環境変数を C ロケールか無効にする。あるいは [A-Za-z0-9] などのようにして置換対象の半角文字を具体的に指定するのでもよい。

```
$ echo 'MSX ＭＳＸ２　ＭＳＸ２＋'| sed 's/[[:alnum:]]/*/g' ↵
*** **** ****＋                          ←ロケールが日本語設定になっているとこうなってしまう
$ echo 'MSX ＭＳＸ２　ＭＳＸ２＋'| LANG=C sed 's/[[:alnum:]]/*/g' ↵  ← C ロケールにする
*** ＭＳＸ２　ＭＳＸ２＋
$ echo 'MSX ＭＳＸ２　ＭＳＸ２＋'| env -i sed 's/[[:alnum:]]/*/g' ↵  ←環境変数を無効化
*** ＭＳＸ２　ＭＳＸ２＋
$ echo 'MSX ＭＳＸ２　ＭＳＸ２＋'| sed 's/[A-Za-z0-9]/*/g' ↵    ←明確に半角英数字を指定
*** ＭＳＸ２　ＭＳＸ２＋
$
```

解説

ロケール環境変数（LC_*や LANG）を認識してくれる **GNU 版の grep や sed、AWK コマンドの正規表現は、文字クラス（[[:alnum:]] や [[:blank:]] など）を用いた場合、全角の文字を半角で対応する文字と同一視**する。知っていれば便利だろうが、知らずにそうなってしまった場合は「なんてお節介な!」と思いたくなる仕様であろう。

無効にする方法は簡単なので、回答で示したとおりにやればよい。しかしそもそも、どの環境でも動くコードを目指すために、

- 意図しない環境変数はシェルスクリプトの冒頭で無効化
- 文字クラスは使わない

をお勧めする。

参照

→レシピ 5.6（ロケール）

レシピ 2.4　sort コマンドの基本と応用とワナ

問題

UNIX の sort コマンドはいろいろな機能があって強力だときいたが、うまく使えない。

回答

確かに UNIX の sort コマンドは多機能だ。使いこなせば殆どの要求に応えられるだろう。しかし知らないとハマるワナがいくつかあるし、またこの質問者は基本からおさらいした方がよさそうだ。そこで、sort コマンドチュートリアルを行うことにする。何にもオプションを付けずに `sort` と打ち込むくらいしか知らないというなら、これを読んで便利に使おう。

基本編. 各行を単なる 1 つの単語として扱う

sort コマンドの使い方には基本と応用がある。基本的な使い方は単純で、**各行を 1 つの単語のように見なしてキャラクターコード順に並べるなどの使い方だ。**

■(オプションなし)……キャラクターコード順に並べる

```
$ cat <<EXSAMPLE | sort ↵
> perl ↵
> ruby ↵
> Perl ↵
> Ruby ↵
> EXSAMPLE ↵
Perl                ← 註)
Ruby                ← キャラクターコード順なので
perl                ← 大文字から先に並ぶ
ruby                ←
$
```

■-f……辞書順に並べる

```
$ cat <<EXSAMPLE | sort -f ↵
> perl ↵
> ruby ↵
> Perl ↵
> Ruby ↵
> EXSAMPLE ↵
Perl                ← 註)
perl                ← 辞書順なので
Ruby                ← P,p,R,r の順で
ruby                ← 並ぶ
$
```

■-n……整数順に並べる

```
$ cat <<EXSAMPLE | sort -n ↵
> 2 ↵
> 10 ↵
> -3 ↵
> 1 ↵
> EXSAMPLE ↵
-3                  ← 註)
1                   ← 値の小さい順に並ぶ。
2                   ← もし-n を付けないと
10                  ← -3,1,10,2 の順に並ぶことになる (2 と 10 の順番が狂ってしまう)。
$
```

※ マイナス記号は認識するが、プラス記号は認識しない。

■-g……実数順に並べる (POSIX 非標準)

```
$ cat <<EXSAMPLE | sort -g ↵
> +6.02e+23 ↵
> 1.602e-19 ↵
> -928.476e-26 ↵
> EXSAMPLE ↵
-928.476e-26        ← 註)
1.602e-19           ← 浮動小数点表記でも正しくソートする。
+6.02e+23           ← -n オプションと違い、+ 符号も認識する。
$
```

※ 単純な整数にも使える、計算量が多くなるので、整数には-n オプションの方がよい。

■-r……降順に並べる (他オプションと併用可)

```
$ cat <<EXSAMPLE | sort -gr ↵
> +6.02e+23 ↵          ↑註 1) 他のオプションと組み合わせて使える
> 1.602e-19 ↵
> -928.476e-26 ↵
> EXSAMPLE ↵
+6.02e+23          ← 註 2)
1.602e-19          ← 先程の-g オプションとは、
-928.476e-26       ← 順番が正反対になっている。
$
```

応用編. 複数の列から構成されるデータを扱う

sort コマンドの本領は、ここで紹介する使い方を覚えてこそ発揮される。SQL の “ORDER BY” 句のように、第 1 ソート条件、第 2 ソート条件……、と指定できるのだ。強力である。

応用編では、2 つのサンプルデータを例に紹介する。

サンプルデータ (1)…キャラクター名簿

次のように、キャラクター初出年、性別、キャラクター名、の 3 列から構成される半角スペース区切りのデータ（sample1.txt）があったとしよう。

■sample1.txt

```
1992 男 りょうおうき
1992 女 まさきささみじゅらい
2003 女 かわはらさき
2003 男 しらせあきら
1995 男 るみや
1995 女 あまのみさお
```

ちなみに列と列の間の半角スペースは 1 つでなければならない。2 つのままだとデータによっては失敗するのだが、それについては「ワナ編」で説明しよう。

さてここで「名前順にソートせよ」という要請を受けたとする。名前が 1 列目にあれば簡単（単にオプション無しの sort に渡すだけ）なのだが、このサンプルデータでは 3 列目にある。こういう時は、-k 3,3 というオプションを付けてやる。つまり、-k オプションの後ろに**ソートしたい列番号をカンマ区切りで 2 つ書く**。

```
$ sort -k 3,3 sample1.txt ↵
1995 女 あまのみさお
2003 女 かわはらさき
2003 男 しらせあきら
1992 女 まさきささみじゅらい
1992 男 りょうおうき
1995 男 るみや
$
```

なぜ 2 つ付けるのかについてであるが、入門段階ではとりあえず「そういうもんだ」と思って覚えておけばよい[*4]。

さて次に「名前を降順にソートせよ」という要請を受けたとする。先程は名前をキャラクターコードの昇順にソートしたが、逆順にしたい場合はどうするか。答えは`-k 3r,3` である。つまり、最初の数字の直後に**基本編で紹介したオプション文字を付ける**。これもとにかくそういうものだと覚えておけばよい。

```
$ sort -k 3r,3 sample1.txt ↵
1995 男 るみや
1992 男 りょうおうき
1992 女 まさきささみじゅらい
2003 男 しらせあきら
2003 女 かわはらさき
1995 女 あまのみさお
$
```

ちなみに、もし名前が半角アルファベットで記述されていて、それをアルファベット順に並べたかったとするなら、`-k 3fr,3` と書けばよい。`f` は基本編で出てきた「辞書順に並べる」オプションだ。

今度は「性別→初出年逆順→名前の順でソートせよ」という要請を受けたとしよう。複数のソート条件を指定する場合にはどうすればいいか。答えは「`-k` オプションを複数書く」である。つまりこの場合、`-k 2,2 -k 1nr,1 -k 3,3` だ。

```
$ sort -k 2,2 -k 1nr,1 -k 3,3 sample1.txt ↵
2003 女 かわはらさき
1995 女 あまのみさお
1992 女 まさきささみじゅらい
2003 男 しらせあきら
1995 男 るみや
1992 男 りょうおうき
$
```

性別を昇順にすると「女」が先に来るのは、男女を音読みした場合の名前順で女の方が先だからである。ま

[*4] どうしても詳しく知りたい人は FreeBSD や Linux の man ページの sort(1) を見るとよいだろう。

た、初出年の降順ソート（-k 1nr,1）で “n” を付けているのは、もし初出年が 3 桁以下だった場合でも正常に動作することを保証するためだ。

サンプルデータ (2)…パスワードファイル

ここでサンプルデータを変える。/etc/passwd ファイルだ。誰もが持ってるので試すのも楽だろう。中を覗いてみるとこんな感じになっているはずだ。

■/etc/passwd の例

```
# $FreeBSD: release/9.1.0/etc/master.passwd 218047 2011-01-28 22:29:38Z pjd $
#
root:*:0:0:Charlie &:/root:/bin/csh
toor:*:0:0:Bourne-again Superuser:/root:
daemon:*:1:1:Owner of many system processes:/root:/usr/sbin/nologin
operator:*:2:5:System &:/:/usr/sbin/nologin
bin:*:3:7:Binaries Commands and Source:/:/usr/sbin/nologin
tty:*:4:65533:Tty Sandbox:/:/usr/sbin/nologin
kmem:*:5:65533:KMem Sandbox:/:/usr/sbin/nologin
 :
```

特徴は、スペース区切りではなくてコロン区切りになっている点だ。あと、先頭にコメント行がついているが、これは正しくソートできないのでソートの直前には grep -v ’^#’ を挿んで取り除かなければならない。

ここで次の要請「グループ番号→ユーザー番号の順でソートせよ」を受けたとする。ソート例は一つしかやらないが、sample1.txt を踏まえれば-k オプションを使ってどうやるかについてはもうわかるはずだ。グループ番号は第 4 列、ユーザー番号は第 3 列にあるのだから-k 4n,4 -k 3n,3 とすればよい。

問題は区切り文字だ。**列と列を区切る文字が半角スペース以外の場合には-t オプションを使う。**/etc/passwd はコロン区切りなので-t ’:’ と書く。

まとめると答えはこうだ。

```
$ cat /etc/passwd              | ↵
> grep -v ’^#’                 | ↵←コメント行を除去する
> sort -k 4n,4 -k 3n,3 -t ’:’ ↵
root:*:0:0:Charlie &:/root:/bin/csh
toor:*:0:0:Bourne-again Superuser:/root:
daemon:*:1:1:Owner of many system processes:/root:/usr/sbin/nologin
unbound:*:59:1:unbound dns resolver:/nonexistent:/usr/sbin/nologin
operator:*:2:5:System &:/:/usr/sbin/nologin
  :
kmem:*:5:65533:KMem Sandbox:/:/usr/sbin/nologin
nobody:*:65534:65534:Unprivileged user:/nonexistent:/usr/sbin/nologin
$
```

ワナ編. 区切り文字に潜むワナ 2 つ

おまちかねのワナ編。応用編のテクニックを使いこなすには、この 2 つのワナも覚えておかないとハマることになる。

その 1 半角スペース複数区切りのワナ

また新たなサンプルデータファイルを用意する。

■sample2.txt

```
1  B -
10 A -
```

ご覧のように第 2 列の位置を揃えるために、1 行目の文字 “B” の手前には半角スペースが 2 個挿入されている。このような例は、df、ls -l、ps などのコマンド出力結果や、fstab などの設定ファイルで身近に溢れている。

このデータを第 2 列のキャラクターコード順にソートしたらどうなるか？ 1 行目と 2 行目が入れ替わってもらいたいところだが、やってみると入れ替わらないのだ。少なくとも GNU sort 8.4 で実行したらそうなった。

```
$ sort -k 2,2 sample2.txt ↵
1  B  -
10 A  -
$
```

理由は、AWK コマンド等と違い、**sort コマンドは複数の半角スペースを 1 つの列区切りと見なせない**からなのだ。1 行目の 1 つ目と 2 つ目の半角スペースの間にヌル文字から成る第 2 列が存在すると見なしているのである。

従って列と列を区切る複数のスペースは 1 個にしなければならない。だから、上記のコードは下記のように修正すれば正しく動く。

```
$ cat sample2.txt                       | ↵
> sed 's/[[:blank:]][[:blank:]]*/ /g' | ↵←註 1) 連続するスペースを 1 つにする
> sed 's/^[[:blank:]]*//'             | ↵←註 2) 行頭のスペースを除去する
> sed 's/[[:blank:]]*$//'             | ↵←註 3) 行末のスペースを除去する
> sort -k 2,2 ↵
10 A  -
1 B  -
$
```

まぁ当然、位置取りのスペースが消えてガタガタになってしまうのだが……。

あと、2 個目と 3 個目の sed もあった方が安全だ。これは、ps コマンドのように行頭 (第 1 列の手前) にも半角スペースを入れる場合のあるコマンドで誤動作しないようにするための予防策である。

その 2 全角スペース区切りのワナ

ロケールに関する環境変数（LC_*、LANG など）が設定してある環境で使っている人にはもう一つのワナが待ち構えているので注意しなければならない。

次のサンプルデータを見てもらいたい。これは、第 1 列に人名、第 2 列にかな、という構成の名簿データだ。注目すべきは苗字と名前の間には全角スペースが入っている点である。

■sample3.txt

```
白瀬　慧 しらせ　あきら
天野　美紗緒 あまの　みさお
本木　紗英 もとき　さえ
```

このデータを名簿順にソートせよと言われたとする。普通に考えれば、`-k 2,2` でいいはずだ。読みがなが第 2 列にあるのだから。ところが日本語ロケール (`LANG=ja_JP.UTF-8` など) になっている Linux 環境で実行すると失敗する。

```
$ sort -k 2,2 sample3.txt ↵
白瀬　慧 しらせ　あきら
天野　美紗緒 あまの　みさお
本木　紗英 もとき　さえ
$
```

原因は、**全角スペースも列区切り文字扱い**されているということだ。正しくやるには、環境変数を無効にする、もしくは**-t オプションで区切り文字は半角スペースだと設定**しなければならない。

具体的には、sort コマンド引数に`-t ' '`と追記してやればよい。

```
$ sort -k 2,2 -t ' ' sample3.txt ↵
天野　美紗緒 あまの　みさお
白瀬　慧 しらせ　あきら
本木　紗英 もとき　さえ
$
```

おめでとう、これでアナタも今日から sort コマンドマスターだ。本当はここで説明していない機能が他にもあるが、それらについて知りたければ、使っている OS の man コマンドで sort について調べてもらいたい。

参考

→レシピ 1.10（ブラックリストの 100 件を 1 万件の名簿から除去する）…join コマンドの話

レシピ 2.5　sed の N コマンドの動きが何かおかしい

問題

手元の FreeBSD 環境（9.1-RELEASE）で下記の sed コマンドを実行したのだが、何だか挙動がおかしかった。

```
seq 1 10 | sed '3,4N; s/¥n/-/g'
```

回答

確かにヘンな動きをする。GNU 版ではこの問題は起きないし、最新版（10.1-RELEASE）では解消されているのでどうやらバグのようだ。バージョンアップまたは GNU 版の使用をお勧めする。ただし、GNU 版は

GNU 版でまた別のヘンな動きをする。

解説

問題文で示された sed コマンドは「元データの 3 行目に関しては、読み込んだら次の行も追加で読み込み、その際残った改行コードを “-” に置換してから出力せよ」という意味である。もっとわかりやすく言えば「3 行目と 4 行目はハイフンで繋げ」という意味である。従って、正常な動作であるなら次のようになるはずである。

```
$ seq 1 10 | sed '3,4N; s/¥n/-/g' ↵
1
2
3-4
5
6
7
8
9
10
$
```

ところが、FreeBSD 9.1-RELEASE の sed では次のようになってしまう。

```
$ seq 1 10 | sed '3,4N; s/¥n/-/g' ↵
1
2
3-4
5-6     ←これは
7-8     ←おかしい！
9
10
$
```

この問題はその後、修正コードと共にバグとして報告され、執筆時の最新版である FreeBSD 10.1 では修正されている。従って、9.x や 10.x を使っているなら OS を最新版にアップグレードすることをお勧めする。もしそれが難しいのであれば GNU 版を使うこともやむを得ないだろう。

GNU 版にも別のバグが

ただし GNU 版にも同じ N コマンドでまた別のバグが見つかっている。

GNU 版の独自拡張である、行番号の相対表現を使うと……

```
$ seq 1 10 | gsed '3,+3N; s/¥n/-/g' ↵
1
2
3-4
5-6
7-8     ←これはおかしい！
9
10
$
```

やはりおかしい。3 行目から +3 行目までだから 7 行目と 8 行目が結合されてはいけないはずだ。ちなみにこれは執筆時に取得できた最新版（4.2.2）でも残っていた。

sed で N コマンドを使っているソースコードでもしおかしな動きをしていたら、sed を疑ってみると原因が見つかるかもしれない。

レシピ 2.6　標準入力以外から AWK に正しく文字列を渡す

問題

AWK に値を渡したいのだが、-v オプションで渡しても、シェル変数を使ってソースコードに即値を埋め込んでも一部の文字が化けてしまう。どうすればよいか。ただし、標準入力は他のデータを渡すのに使っており、使えない。

```
str='¥n means "newline"'          ←渡したい文字列

awk -v "s=$str" 'BEGIN{print s}'  ←¥n がうまく渡せない

awk 'BEGIN{print "'"$str"'"}'     ←¥n も"〜"もうまく渡せない（エラーにもなる）
```

回答

環境変数として渡し、AWK の組込変数 ENVIRON で受け取る。問題文の “¥n means "newline"” を渡したいのであれば、こう書けばよい。

```
str='¥n means "newline"' awk 'BEGIN{print ENVIRON["str"]}'
```

既にシェル変数に入っているのであれば export して環境変数化してから渡してもいいし、それが嫌ならコマンドの前に仮の環境変数（例えば “E”）を置いて渡したり、env コマンドで渡してもいいだろう。

```
str='¥n means "newline"'

export str
awk 'BEGIN{print ENVIRON["str"]}'

E=str awk 'BEGIN{print ENVIRON["E"]}'

env E=str awk 'BEGIN{print ENVIRON["E"]}'
```

解説

何らかの事情で AWK に値を渡したいとなったら、手段はいくつかある。

1. -v オプションで AWK 内の変数を定義して渡す
2. AWK のコードに埋め込んで即値として渡す
3. 標準入力から渡す
4. 環境変数として渡す

1 番目は定番で、2 番目も（筆者は）よくやる方法だ。だが、バックスラッシュを含む文字が化けるという問題がある。2 番目に関しては問題文に示したように、ダブルクォーテーションを含む場合に単純な文字化けでは済まず、セキュリティーホールを生みかねない誤動作を招く。従ってこれらの方法はどんな文字が入っているかわからない文字列を渡すのには使えない。3 番目の標準入力を使えれば安全なのだが、メインのデータを受け取るために既に使用中という場合もある。そうなると残る選択肢が 4 番目の環境変数というわけだ。

AWK は起動直後、環境変数を “ENVIRON” という名の組込変数に格納してくれる。これは連想配列なので環境変数名をキーにして読み出す。通常はこの変数によって現在設定されているロケールやコマンドパスを知るのに使うところだが、もちろんユーザーが自由に環境変数を定義してもよい。しかも都合の良いことに、一切エスケープされることなく伝わる。例えばこのようにして、シェルの組込変数 IFS[*5]を伝えることもできる。

```
$ ifs="$IFS" awk 'BEGIN{print "(" ENVIRON["ifs"] ")",length(ENVIRON["ifs"]) }' ↵
(           ← 括弧の中に空白、タブ、改行が表示され、
) 3         ← その直後に変数のサイズ（文字数）が 3 であると表示された。
$
```

どんな文字が入っているかわからない文字列を安全に渡したい場合に知っておきたい手段だ。

[*5] 文字列の列区切りと見なす文字列を定義しておく環境変数。for 構文等で参照される。デフォルトでは半角スペースとタブ、そして改行コードが入っている。

レシピ 2.7　AWK の連想配列が読むだけで変わるワナ

問題

AWK の配列で、必要な要素 “3” がきちんと生成できていないことが原因で中断している疑いのあるコードがあった。そこで問題の要素 “3” に確実に値が入っているかどうかを確認するデバッグコードを入れて動かしたところ、中断せずに動くようになってしまった。もしかして AWK のバグか??

```
awk 'BEGIN{
  str = "data(1/3) data(2/3)";                  # ← 1) 本来あるべき第三列"data(3/3)"が無い
  split(str, ar);
print "***DEBUG*** array#3:",ar[3];             # ← 3) ここにデバッグ用コードを入れたら
  if ((1 in ar) && (2 in ar) && (3 in ar)) {    #       エラー終了しなくなってしまった!
    print "#1:", ar[1];
    print "#2:", ar[2];
    print "#3:", ar[3];
  } else {
    print "データが足りません" > "/dev/stderr";   # ← 2) 冒頭の問題により
    exit;                                       #       ここでエラー終了してしまっていた
  }
}'
```

回答

AWK は、存在しない配列要素を読み込むと、その時点で空の要素が生成する。これは AWK の仕様であるので気を付けなければならない。

配列 “*array*” の要素 “*key*” の内容を確認するコードの前には、“(*key* in *array*)” などと記述して、まずその要素が存在していることを確認すること。

解説

「回答」にも記したが、AWK の配列変数は、存在しない要素を読み込むと、空文字を値として勝手にその要素を作成してしまう。bash 等、他の言語の配列変数ではこのようなことはないのだが、AWK ではこのように動作することが仕様であり、バグでない。従って、そういうものだと覚えるしかない。

それではもう一つ例を見てみよう。次のシェルスクリプトを書いて、実行してみてもらいたい。

■awk_test.sh

```
#! /bin/sh

awk '
BEGIN{
  split("", array);    # 連想配列を初期化（要素数 0 にする）
  print length(array); # 要素数は当然"0"と表示される。

  print array["hoge"]; # だから"hoge"なんて要素を表示しようとしても当然空行

  print length(array); # ところがもう一度要素数を見てみると……
}

# 配列に対する length に非対応の AWK 実装を使っている場合は
# 下記のコードも記述したうえで、上記の length を全て arlen に書き換えて実行する
function arlen(ar,i,l){for(i in ar){l++;}return l;}
'
```

実行するとこうなるはずだ。

```
$ ./awk_tesh.sh ↵
0


1       ←要素数が 1 になっている
$
```

AWK の配列変数の取扱いにはご注意を。

レシピ 2.8　while read で文字列が正しく渡せない

問題

操作ミスで変な名前のファイル名がいっぱいできた時にそれらを削除するワンライナーを書いたが、一部のファイルは “No such file or directory” となって消せずに残ってしまう。なぜか。

```
ls -1a "$dir" | grep -Ev '^¥.¥.?$' | while read file; do rm "$file"; done
```

回答

このワンライナーが、read コマンドの使用上の注意（下記の 2 つ）を見逃しているからである。

- -r オプションを付けない場合、read コマンドはバックスラッシュ “¥” をエスケープ文字扱いする。
- read コマンドは行頭、行末にある半角スペースとタブの連続を除去する。

この仕様を回避するため、ワンライナーは次のように直す必要がある。

```
ls -1a "$dir" | grep -Ev '^¥.¥.?$' | sed 's/^/_/' | sed 's/$/_/' | while read -r file; do file=${file#_}; file=${file%_}; rm "$file"; done
```

解説

1 行ごとに処理をする時の定番である “while read” 構文。標準入力からパイプを使って while read ループにテキストデータを渡す処理を書いた場合、その中で書き換えたシェル変数はループの外には反映されないという落とし穴があるのは有名になってきた。しかし、油断するとループ内にデータを正しく渡せないという落とし穴にも気を付けなければならない。

その落とし穴の具体的な内容については「回答」で書いたが、さてその対策として示したワンライナーは何をやっているのだろうか。

-r オプションを付ける

一つ目のこれは単純だ。バックスラッシュ “¥” がエスケープ文字扱いされぬように、read コマンドに-r オプションを追加すればそれでおしまいである。

文字の前後にダミー文字を付加、ループ内で除去

二つ目の対策はワンライナーに 4 ステップを追加しているのでちょっと複雑かもしれない。read コマンドは行頭と行末のスペース類（半角スペースとタブの連続）を取り除くのだから、予めそうでないものを追加しておいてそれを回避しようとしているのだ。そのために、各行の行頭・行末にアンダースコアを追加する 2 の sed を追加し、ループの中で、それらを除去する変数トリミング（シェル変数の中にある “#” と “%”）処理を追加してある。

レシピ 2.9　あなたはいくつ問題点を見つけられるか!?

問題

次のシェルスクリプトは引数で指定したディレクトリー直下にあるデッドリンク（実体ファイルを失ったシンボリックリンク）を見つけて削除するためのものである。しかし、いくつも問題点を含んでいると指摘された。優秀な読者の皆さんに問題点を全部指摘してもらいたい。

```
#! /bin/sh

[ $# -eq 1 ] || {
  echo "Usage : ${0###*/} <target_dir>" 1>&2
  exit 1
}

dir=$1

cd $dir
ls -1 |
while read file; do
  # デッドリンクの場合、"-e"でチェックすると偽が返される
  [ -L "$file" ] || continue
  [ -e  $file  ] || rm -f $file
done
```

概要

これは本章のまとめとしての、演習問題だ。まとめといっても本章では説明していないこともあるが……。さて読者の皆さん、全部見つけられるかな？:-)

:
:
:

解答

1. 意図しない場所の同名コマンドが実行される恐れがある

環境変数 PATH がいじくられていると同名の予期せぬコマンドが実行される恐れがある。安全を期すなら、環境変数 PATH を/bin と/usr/bin だけにすべきだ。今回使っているコマンドはいずれも POSIX 範囲内なので、どちらかのディレクトリーにあるはずだ。

■環境変数 PATH がヘンに弄られていた場合への対策

```
PATH=/bin:/usr/bin # シェルスクリプトの冒頭にこの行を追加
```

尚、「/bin や/usr/bin にあるコマンドそのものが不正に書き換えられていたら？」という指摘があるかもしれないが、それは OS そのものが既に正常ではないことを意味し、言いだしたらキリがないためここでは無視する。

2. 引数$1 がディレクトリーであることを確かめていない

ディレクトリーでない引数を指定するとその後の cd コマンドが誤作動する。そのため冒頭の test コマンド ([) に、引数がディレクトリーとして実在していることを確認するためのコードを追加すべきである。

■ディレクトリーの実在性確認

```
[ ¥( $# -eq 1 ¥) -a ¥( -d "$1" ¥) ] || {     # ディレクトリー実在性確認を追加
  echo "Usage : ${0##*/} <target_dir>" 1>&2
  exit 1
}
```

3. 引数$1 が-で始まっていると誤作動する

ディレクトリー名がハイフンで始まっていると、cd コマンドにそれはオプションであると誤解され、誤動作してしまう。これを防ぐためには、絶対ディレクトリーでないと判断した時、強制的に先頭にカレントディレクトリー./を付けるようにすべきである。

具体的には、シェル変数 dir を代入している行の後ろに次のコードを追加する。

■ディレクトリー名がハイフンで始まる場合への対策

```
case "$dir" in
  /*) :;;              # 先頭が/で始まってる（絶対パス）ならそのまま。
  *)  dir="./$dir";;   # さもなければ先頭に"./"を付ける。
esac
```

4. 引数$1 がスペースを含んでいると誤動作する

半角スペースを含んでいるような特殊なディレクトリー名だと、cd コマンドには複数の引数として渡され、誤動作してしまう。そうならないように cd コマンドの引数$dir はダブルクォーテーションで囲むべきである。

■ディレクトリー名がスペースを含む場合への対策

```
cd "$dir"
```

5. 引数$1 のディレクトリーに移動できなかった場合でも作業が止まらない

いくら引数$1 でディレクトリーが実在していることを確認しても、パーミッションが無い等の理由でそのディレクトリーに移動できなかったら……。そう、意図せずカレントディレクトリーのデッドリンクを消そうとしてしまうのだ。そうなってしまわないように cd コマンドの次の行に下記のコードを挿入し、ディレクトリー移動に失敗したら、中断するようにすべきである。

■指定されたディレクリーに移動できなかった場合への対策

```
[ $? -eq 0 ] || exit 1
```

6. 隠しファイルを見逃してしまう

UNIX では先頭がピリオド “.” で始まるファイルは、（特殊ファイルの “.” と “..” を除き）隠しファイルとして扱われる。従って ls コマンドでもデフォルトでは隠しファイルを列挙しないが、これでは隠しファイルとして存在するデッドリンクも見つけられない。隠しファイルでも列挙するようにするため、ls コマンドには-a オプションを追加すべきである。

■ls コマンドに隠しファイルを列挙させるための対策

```
ls -1a |
```

7. ファイル名がスペースを含んでいると誤動作する

これも指摘 4 と同じ理屈だ。test コマンドも rm コマンドも誤動作してしまう。よって両方ともダブルクォーテーションで囲むべきである。

■ファイル名がスペースを含んでいる場合への対策

```
  [ -L "$file" ] || continue
  [ -e "$file" ] || rm -f "$file"
```

8. ファイル名が-で始まっていると誤作動する

これも指摘 4 と同じ理屈だ。もし、先の 6 の対策コードも下記に記す対策コードもなかったら、“-rf /home/your_homedir” などというヒネくれたリンクファイルが置かれていた日には、大変なことになるぞ！

■ファイル名がハイフンで始まる場合への対策

```
  file="./$file"
```

9. ファイル名にバックスラッシュを含んでいるものを正しく扱えない

これはレシピ 2.8（while read で文字列が正しく渡せない）で示した問題だ。read コマンドはデフォルトだとバックスラッシュをエスケープ文字扱いするため、そうさせないように read コマンドには-r オプションを追加

すべきである。

■ファイル名がハイフンで始まる場合への対策

```
while read -r file; do # -r オプションを追加する
```

10. ファイル名の先頭・末尾にスペース類が付いているものを正しく扱えない

これもレシピ 2.8（while read で文字列が正しく渡せない）で示した問題だ。read コマンドは文字列の先頭・末尾に付いているスペース類（半角スペースとタブの連続）を除去してしまうから、文字列の両端にそうでない文字を付加してから read コマンドを通し、抜けたところで直ちに除去すべきである。

■ファイル名がハイフンで始まる場合への対策

```
ls -1a       |          # （-a オプションは指摘 6 での対策）
sed 's/^/_/' |          # ファイル名の先頭にダミー文字として"_"を付加
sed 's/$/_/' |          # ファイル名の末尾にダミー文字として"_"を付加
while read -r file; do # （-r オプションは指摘 8 での対策）
  file=${file#_}        # ファイル名の先頭につけたダミー文字"_"を除去
  file=${file%_}        # ファイル名の末尾につけたダミー文字"_"を除去
```

まとめ

以上、指摘された 10 項目を全て反映させ、修正すると次のとおりになる。

```
#! /bin/sh

PATH=/bin:/usr/bin

[ ¥( $# -eq 1 ¥) -a ¥( -d "$1" ¥) ] || {
  echo "Usage : ${0##*/} <target_dir>" 1>&2
  exit 1
}

dir=$1
case "$dir" in
  /*) :;;
  *)  dir="./$dir";;
esac

cd "$dir"
[ $? -eq 0 ] || exit 1
ls -1a |
sed 's/^/_/' |
sed 's/$/_/' |
while read -r file; do
  file=${file#_}
  file=${file%_}
  file="./$file"
  # デッドリンクの場合、"-e"でチェックすると偽が返される
  [ -L "$file" ] || continue
  [ -e "$file" ] || rm -f "$file"
done
```

果たして、全部指摘することはできただろうか……。何、「これ以外にも指摘がある」と!? それは是非、筆者に教えてもらいたい！

第 3 章

POSIX 原理主義テクニック

「一体 POSIX の範囲で何ができる？」と言っているそこのアナタ。POSIX を見くびるのはこの章を読んでからにしてもらおうか。たくさんのコマンドに支えられているおかげで、POSIX の範囲でも実にいろいろなことができる。そもそも、そんなコマンドの一つである AWK や sed はチューリングマシンの要件を満たしているのだから、入出力がファイルの世界で閉じている作業であれば何でもできるのである。

というわけで本章では、POSIX の範囲で仕事をこなす様々なテクニックを紹介する。**機能を求めて他言語に手を出すなど百年早い！**

レシピ 3.1　PIPESTATUS さようなら

問題

bash 上で動いていたシェルスクリプトを、他のシェルでも使えるように書き直している。しかし、組込変数の PIPESTATUS を参照している箇所があり、これを書き換えられずに悩んでいる。PIPESTAUTS 相当の変数を用意する方法はないか？

回答

方法はあるので安心してもらいたい。

まず、次に示すシェル関数 “run()” をシェルスクリプトの中で定義する。

■PIPESTAUTS 相当の機能を実現するシェル関数 “run()”

```
run() {
  local a j k l com # ←ここは POSIX 範囲外なんだけど……
  j=1
  while eval "¥${pipestatus_$j+:} false"; do
    unset pipestatus_$j
    j=$(($j+1))
  done
  j=1 com= k=1 l=
  for a; do
    if [ "x$a" = 'x|' ]; then
      com="$com { $l "'3>&-
                  echo "pipestatus_'$j'=$?" >&3
                } 4>&- |'
      j=$(($j+1)) l=
    else
      l="$l ¥"¥${$k}¥"" # ←修正箇所はここ
    fi
    k=$(($k+1))
  done
  com="$com $l"' 3>&- >&4 4>&-
             echo "pipestatus_'$j'=$?"'
  exec 4>&1
  eval "$(exec 3>&1; eval "$com")"
  exec 4>&-
  j=1
  while eval "¥${pipestatus_$j+:} false"; do
    eval "[ ¥$pipestatus_$j -eq 0 ]" || return 1
    j=$(($j+1))
  done
  return 0
}
```

そして、この “run” を頭に付ける形で、パイプに繋がれた一連のコマンドを実行する。ただし、このシェル関数に引数を渡すため、シェルが解釈してしまう文字は全て、エスケープするかシングルクォーテーション等で囲むこと。（詳細は「解説」を参照）

```
run command1 ¥| command2 '2>/dev/null' ¥| ...
```

各コマンドの戻り値は、“pipestatus_n”（n はコマンドの順番で、最初は 1）に格納されている。

解説

このシェル関数はもともと Web 上で公開されてるもの[*1]である。

しかしながら、引数が 10 個以上になると動作しなくなる不具合を抱えているために若干の修正を加えた（2 番目のコメント部分）。また、シェル関数内で使われている変数をローカルスコープにするため、関数の冒頭で local 宣言をしているが、これは POSIX 範囲を逸脱しているため、使えなければ外しつつ、中で使っている 5 つのシェル変数に気を付けること。

[*1] The UNIX and Linux Forums の “return code capturing for all commands connected by "|" ...” というスレッドである。URL は http://www.unix.com/302268337-post4.html だ。

実際に使ってみる

ここで試してみるコマンドは次のものとする。

```
printf 1                        |
awk '{print $1+1}END{exit 2}' |
cat                             |
awk '{print $1+1}END{exit 4}' |
cat
```

動作シナリオはこうだ。1 行目で値 “1” を渡され、2 行目と 4 行目の AWK を通るたびに 1 つ加算され、最後は “3” と表示される。ただし、途中の AWK は戻り値としてそれぞれ 2 と 4 を返す。もし PIPESTATUS が使えるなら、先頭から順に、0、2、0、4、0 という戻り値が得られるわけだ。

さて、先のシェル関数 run() を使って前述のコマンドを書き直したシェルスクリプトを用意してみる。

■run() 関数のテスト用シェルスクリプト pipestatus_test.sh

```
#! /bin/sh

# 1) シェル関数 run() の定義
run() {
  :       # ここに、前述のシェル関数 run() の中身を書く
}

# 2) run() を使って実行する
run                               ¥
printf 1                        ¥| ¥
awk '{print $1+1}END{exit 2}' ¥| ¥
cat                             ¥| ¥
awk '{print $1+1}END{exit 4}' ¥| ¥
cat

# 3) pipestatus の内容を列挙してみる
set | grep '^pipestatus_'
```

run() を使った書き方に注意。このコードのように、可読性確保のために改行をさせている場合は行末にバックスラッシュを付けておかねばならない。この時点で注意すべき事をまとめるとこうだ。

- パイプで繋ぐ一連のコマンド列の先頭にキーワード “run” をつける。
- コマンドを繋ぐパイプ記号 “|” はエスケープする。
- その他、シェルにエスケープされては困る文字（| はもちろん、&、>、<、(、)、{、}など）も全てエスケープする、あるいはシングルクォーテーションで囲む。
- 可読性のために改行を入れたい場合は、行末にバックスペース “¥” を付けることによって行う。

書き終えたら実行してみよう。

```
$ sh pipestatus_test.sh ↵
3
pipestatus_1=0
pipestatus_2=2
pipestatus_3=0
pipestatus_4=4
pipestatus_5=0
$
```

各コマンドの戻り値をきちんと拾えていることがわかる。

シェル関数を使わないこともできる

もしシェル関数を使いたくないということであれば、それもできないわけではない。その場合は、run() 関数を使って書いたシェルスクリプトを実行ログが出力される形[*2]で実行してみるとよい。

すると、eval している箇所が見るつかるはずだ。上記のシェルスクリプトで例を示すとこうなる。

```
$ sh -x pipestatus_test.sh ↵     ←-x オプションを付けて実行
   :
+ eval '{  "${1}" "${2}" 3>&-
                  echo "pipestatus_1=$?" >&3
               } 4>&-| {  "${4}" "${5}" 3>&-
                  echo "pipestatus_2=$?" >&3
               } 4>&-| {  "${7}" 3>&-
                  echo "pipestatus_3=$?" >&3
               } 4>&-| {  "${9}" "${10}" 3>&-
                  echo "pipestatus_4=$?" >&3
               } 4>&-|  "${12}" 3>&->&4 4>&-
            echo "pipestatus_5=$?"'
   :
```

run() コマンドはこのようにして、シェルスクリプトを動的に生成して実行しているに過ぎない。だからこれを参考にして自分で作ってしまえばいいのだ。そして、作ったものが次のシェルスクリプトだ。

[*2] sh コマンドから-x オプション付けて実行する。

```
#! /bin/sh

exec 4>&1
eval "$(
        exec 3>&1
        { printf 1                      3>&-         ; echo pipestatus_1=$? >&3; } 4>&- |
        { awk '{print $1+1}END{exit 2}' 3>&-         ; echo pipestatus_2=$? >&3; } 4>&- |
        { cat                           3>&-         ; echo pipestatus_3=$? >&3; } 4>&- |
        { awk '{print $1+1}END{exit 4}' 3>&-         ; echo pipestatus_4=$? >&3; } 4>&- |
          cat                           3>&- >&4 4>&-; echo pipestatus_5=$?
      )"
exec 4>&-

set | grep '^pipestatus_'
```

ファイルディスクリプターの 4 番を最終的な標準出力の出口に、3 番を “pipestatus_n” 作成のための出口にするという実に巧妙な技を使っている。例えそれの理解が難しかったとしても、どう書けばよいかという規則性は見えてくるのではないだろうか。

レシピ 3.2 Apache の combined 形式ログを扱いやすくする

問題

Apache のログファイル（combined 形式）がある。しかしこれ、単純なスペース区切りのファイルではなく、大括弧（[〜]）やダブルクォーテーションで囲まれている区間は、1 つのフィールドとされている。それゆえ、アクセス日時の列、User-Agent の列など、任意のフィールドを抽出することがとても面倒だ。簡単に取り出せるようにならないものか。

回答

sed コマンドを 4 回、tr コマンドを 2 回通せばできる。通すと、各フィールド内の空白文字がアンダースコアに置換され、フィールド区切りとしての空白だけが残るので、以後 AWK などで簡単に列を抽出することができるようになる。

■Apache ログを正規化するシェルスクリプト apacomb_norm.sh

```
#! /bin/sh

# --- その前に、ちょっと下ごしらえ ---
RS=$(printf '¥036')             # 元々の改行位置を退避するための記号定義
LF=$(printf '¥¥¥n_');LF=${LF%_} # sed で改行コードを挿れるための定義
c='_'                           # ここに空白の代替文字（今はアンダースコアにしている）

# --- 本番 ---
cat "$1"                                    | # 第一引数で Apache ログを指定しておく
sed 's/^¥(.*¥)$/¥1'"$RS"'/'                 |
sed 's/"¥([^"]*¥)"/'"$LF"'"¥1"'"$LF"'/g'    |
sed 's/¥[¥([^]]*¥)¥]/'"$LF"'[¥1]'"$LF"'/g' |
sed '/^["[]/s/[[:blank:]]/'"$c"'/g'         |
tr -d '¥n'                                  |
tr "$RS" '¥n'
```

このシェルスクリプトを通した後、日時フィールドが欲しければ `awk '{print $4}'`、同様に HTTP リクエストパラメーターフィールドなら `awk '{print $5}'`、User-Agent フィールドなら `awk '{print $9}'` をパイプ越しに書き足せばよいわけだ。

解説

ご承知のとおり、Apache で一般的に使われている combined という形式のログはこんな内容になっている。

```
192.168.0.1 - - [17/Apr/2014:11:22:33 +0900] "GET /index.html HTTP/1.1" 200 43206 "https://www.google.co.jp/" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/34.0.1847.116 Safari/537.36"
```

User-Agent フィールド（"mozilla/5.0 (Windows NT 6.1 …… Safari/537.36"の部分）が欲しいと思って、AWK で抽出しようとしても

```
$ awk '{print $12}'httpd-access.log ↵
"Mozilla/5.0
$
```

となってしまって、全然使い物にならない。

しかし、そこは我らが UNIX。シェルスクリプトとパイプと標準コマンドである sed と tr さえあればお手のものだ。他言語に走る必要など全く無い。

「回答」で示したシェルスクリプトに掛けてみるとご覧のとおりだ。

```
$ cat httpd-access.log | apacomb_norm.sh ↵
192.168.0.1 - - [17/Apr/2014:11:22:33_+0900] "GET_/index.html_HTTP/1.1" 200 43206 "h
ttps://www.google.co.jp/" "Mozilla/5.0_(Windows_NT_6.1;_WOW64)_AppleWebKit/537.36_(K
HTML,_like_Gecko)_Chrome/34.0.1847.116_Safari/537.36"
$
```

各々の sed と tr は何をやってるのか？

1 つ 1 つ説明していこう。

■sed #1 （加工の都合により、途中で一時的に改行を挿むので）元の改行を別の文字 <0x1E> で退避させておく。

■sed #2　ダブルクォーテーションで囲まれている区間"〜"があったら、その前後に改行を挿み、その区間を単独の行にする。

■sed #3　ブラケットで囲まれている区間 [〜] も同様に、前後に改行を挿んで、この区間を単独の行にする。

■sed #4　ダブルクォーテーション、またはブラケットで始まる行は、先程行を独立させた区間なので、これらの行にある空白をそうでない文字列に置換する。

■tr #1　改行を全部取り除く。

■tr #2　退避させていた元々の改行を復活させる。

全部 sed でやることもできる

ちなみに tr コマンドを sed に置き換え、全てを sed にすることもできる。

```
cat "$1"                                  |
sed 's/^¥(.*¥)$/¥1'"$RS"'/'               |
sed 's/"¥([^"]*¥)"/'"$LF"'"¥1"'"$LF"'/g'  |
sed 's/¥[¥([^]]*¥)¥]/'"$LF"'[¥1]'"$LF"'/g' |
sed '/^["[]/s/[[:blank:]]/'"$c"'/g'       |
sed 'N;$s/¥n//g'                          |
sed 's/'"$RS"'/'"$LF"'/g'
```

tr コマンドの方が速いので意味のあることではないが……。

コマンド化したものを GitHub にて提供中

いちいち本書を読んで書き写すのも面倒であろうし、少し改良したものを GitHub 上に公開した。よければ使ってみてもらいたい。

https://gist.github.com/richmikan/7254345

スペースの代替文字が_では気に入らない人向けに、オプションで指定できるよにしてある本格派だ。Apache サーバー管理者は、これで少し幸せになれるかもしれない。

参照

→レシピ 1.2（sed による改行文字への置換を、綺麗に書く）

レシピ 3.3　シェルスクリプトで時間計算を一人前にこなす

問題

日常使っている日時（YYYYMMDDhhmmss）と UNIX 時間（UTC 時間による 1970/01/01 00:00:00 からの秒数）の相互変換さえできれば、シェルスクリプトでも日付計算や曜日の算出ができるようになるのだが……。できないものか。

回答

AWK で頑張って実装する。

日常の時間 → UNIX 時間

UNIX 時間への変換は、フェアフィールドの公式から導出される変換式にあてはめるだけなので簡単だ。

■日常の時間 → UNIX 時間 変換シェルスクリプト

```
echo "ここに YYYYMMDDhhmmss" | # date '+%Y%m%d%H%M%S' などを流し込んでもよい
awk '{
  # 年月日時分秒を取得
  Y = substr($1, 1,4)*1;
  M = substr($1, 5,2)*1;
  D = substr($1, 7,2)*1;
  h = substr($1, 9,2)*1;
  m = substr($1,11,2)*1;
  s = substr($1,13  )*1;

  # 計算公式に流し込む
  if (M<3) {M+=12; Y--;} # 公式を使うための値調整
  print (365*Y+int(Y/4)-int(Y/100)+int(Y/400)+int(306*(M+1)/10)-428+D-719163)*86400+(h*3600)+(m*
60)+s;
}'
```

UNIX 時間 → 日常の時間

これは少し複雑だ。一発変換できる公式は無いようだ。そこで glibc の gmtime() 関数を参考に作ったコードを記す。

■UNIX 時間 → 日常の時間 変換シェルスクリプト

```
echo "ここに UNIXtime" |
awk '{
  # 時分秒と、1970/1/1 からの日数を求める
  s = $1%60;  t = int($1/60);  m =  t%60;  t = int(t/60);  h = t%24;
  days_from_epoch = int( t/24);

  # 年を求める
  max_calced_year = 1970;              # To remember every days on 01/01 from
  days_on_Jan1st_from_epoch[1970] = 0; # the Epoch which was calculated once
  Y = int(days_from_epoch/365.2425)+1970+1;
  if (Y > max_calced_year) {
     i = days_on_Jan1st_from_epoch[max_calced_year];
     for (j=max_calced_year; j<Y; j++) {
       i += (j%4!=0)?365:(j%100!=0)?366:(j%400!=0)?365:366;
       days_on_Jan1st_from_epoch[j+1] = i;
     }
     max_calced_year = Y;
  }
  for (;;Y--) {
    if (days_from_epoch >= days_on_Jan1st_from_epoch[Y]) {
      break;
    }
  }

  # 月日を求める
  split("31 0 31 30 31 30 31 31 30 31 30 31", days_of_month);    # 各月の日数 (2 月は未定)
  days_of_month[2] = (Y%4!=0)?28:(Y%100!=0)?29:(Y%400!=0)?28:29;
  D = days_from_epoch - days_on_Jan1st_from_epoch + 1;
  for (M=1; ; M++) {
    if (D > days_of_month[M]) {
      D -= days_of_month[M];
    } else {
      break;
    }
  }

  # 結果出力
  printf("%04d%02d%02d%02d%02d%02d¥n",Y,M,D,h,m,s);
}'
```

解説

シェルスクリプトが敬遠される理由の一つ。それは時間の計算機能が弱いところだろう。例えば、

- 今から一週間前の年月日時分秒は？（それより古いファイルを消したい時など）
- Ya 年 Ma 月 Da 日と Yb 年 Mb 月 Db 日、その差は何日？（ログを整理したい時など）
- この年月日は何曜日？（ファイルを曜日毎に仕分けしたい時など）

といった計算が簡単にはできない。date コマンドの拡張機能を使えばできるものもあるが、できるようになることが中途半端なうえに、OS 間の互換性がなくなる。

前述のような日時の加減算や 2 つの日時の差を求めるなどといった時は、一旦 UNIX 時間に変換して計算し、必要に応じて戻せばよいことはご存知のとおり。曜日を求めるのとて、UNIX 時間変換の値を一日の秒数 (86400) で割って得られた商を、さらに 7 で割って余りを見ればよい、ということもお分かりだろう。

だが、その UNIX 時間との相互変換が面倒だった。そこで変換アルゴリズムを調査した上で POSIX の範囲で実装したものが「回答」で示したコード、というわけである。できないからといって、安易に他言語に頼ろうとする発想は改め、**「無いものは作れ！」**と言っておきたい。自分で作れば理解も深まるし、自由も利く。

コマンド化したものを GitHub にて提供中

いちいち本書を読んで書き写すのも面倒であろうし、少し改良したものを GitHub 上に公開した。よければ使ってみてもらいたい。

```
https://github.com/ShellShoccar-jpn/misc-tools/blob/master/utconv
```

しかもこちらはかなりきっちりやっており、**タイムゾーンを考慮した相互変換**まで対応している。

参照

ちなみに、この時間計算ができるようになると、find コマンドだけでは不十分だったタイムスタンプ比較も自在にできるようになり、シェルスクリプトでも**自力で Cookie が焼けるようになり**、そしてさらに **HTTP のセッション管理ができるようになる。**詳しくはそれぞれのレシピを参照してもたいらい。

→レシピ 3.4（find コマンドで秒単位にタイムスタンプ比較をする）
→レシピ 4.10（シェルスクリプトおばさんの手づくり Cookie(書き込み編)）
→レシピ 4.11（シェルスクリプトによる HTTP セッション管理）

レシピ 3.4　find コマンドで秒単位にタイムスタンプ比較をする

問題

様々な条件でファイルの絞り込みができる find コマンドだが、タイムスタンプでの絞り込み機能が弱い。POSIX 標準では日（=86400 秒）単位でしか絞り込めない。実装によっては分単位まで指定できるものがあるが、独自拡張なのでできたりできなかったりするし記述方法もバラバラだ。

- 指定した年月日時分秒より新しい、より古い、等しい
- n 秒前より新しい、より古い、等しい

という絞り込みはできないものか。

回答

比較用のタイムスタンプを持つファイルを生成し、そのファイルを基準として`-newer` オプションを使えば可能である。

1. 指定日時との比較

日時 “YYYY/MM/DD hh:mm:ss” よりも新しいファイルを抽出したいなら、こんなシェルスクリプトを書けばよい。

```
touch -t YYYYMMDDhhmm.ss thattime.tmp
find /TARGET/DIR -newer thattime.tmp
rm thattime.tmp
```

touch コマンドの書式の事情により、mm と ss の間にピリオドを挿れないといけない点に注意してもらいたい。

次に「より古い」ものを抽出したいならどうするか。それには基準となる日時の 1 秒前（YYYYMMDDhhmms1 とする）のタイムスタンプを持つファイルを作り、-newer オプションの否定形を使えばよい。

```
touch -t YYYYMMDDhhmm.s1 1secbefore.tmp # 基準日時の 1 秒前
find /TARGET/DIR ¥( ¥! -newer 1secbefore.tmp ¥)
rm 1secbefore.tmp
```

それでは「等しい」としたいならどうすればよいか。それには基準日時のファイルとその 1 秒前のファイルの 2 つを作り、「基準日時 1 秒前より新しい」かつ「基準日時を含むそれ以前」という条件にすればよい。

```
touch -t YYYYMMDDhhmm.ss thattime.tmp
touch -t YYYYMMDDhhmm.s1 1secbefore.tmp
find /TARGET/DIR -newer 1secbefore.tmp ¥( ¥! -newer thattime.tmp ¥)
rm 1secbefore.tmp thattime.tmp
```

2. n 秒前より新しい、古い、等しい

基準日時との新旧比較のやり方がわかったのだから、あとは現在日時の n 秒前、および $n-1$ 秒前という計算ができれば実現できることになる。

それはどうやるのかといえば、レシピ 3.3（シェルスクリプトで時間計算を一人前にこなす）を活用すればいい。つまり、日常時間（YYYYMMDDhhmmss）を UNIX 時間に変換して引き算し、逆変換すればいいのだ。こういう需要を想定し、utconv というコマンドは用意されたのである（もちろんシェルスクリプトで）。

それでは、例として 1 分 30 秒前より新しい、等しい、古いファイルを抽出するシェルスクリプトを紹介する。

■1 分 30 秒前より新しいファイルを抽出

```
now=$(date '+%Y%m%d%H%M%S')
t0=$(echo $now                |
     utconv                   |
     awk '{print $0-60*1-30}' |
     utconv -r                |
     sed 's/..$/.&/'          )
touch -t $t0 thattime.tmp
find /TARGET/DIR -newer thattime.tmp
rm thattime.tmp
```

■1 分 30 秒前より古いファイルを抽出

```
now=$(date '+%Y%m%d%H%M%S')
t1=$(echo $now                 |
     utconv                    |
     awk '{print $0-60*1-31}' |
     utconv -r                 |
     sed 's/..$/.&/'           )
touch -t $t1 1secbefore.tmp
find /TARGET/DIR ¥( ¥! -newer 1secbefore.tmp ¥)
rm 1secbefore.tmp
```

■ぴったり 1 分 30 秒前のファイルを抽出

```
now=$(date '+%Y%m%d%H%M%S')
t0=$(echo $now                 |
     utconv                    |
     awk '{print $0-60*1-30}' |
     utconv -r                 |
     sed 's/..$/.&/'           )
t1=$(echo $now                 |
     utconv                    |
     awk '{print $0-60*1-31}' |
     utconv -r                 |
     sed 's/..$/.&/'           )
touch -t $t0 thattime.tmp
touch -t $t1 1secbefore.tmp
find /TARGET/DIR -newer 1secbefore.tmp ¥( ¥! -newer thattime.tmp ¥)
rm thattime.tmp 1secbefore.tmp
```

解説

find コマンドは、様々な条件でファイル抽出ができて便利。でも時間の新旧で絞り込む機能は弱いと言わざるを得ない。

通常のタイムスタンプ (m:ファイルの中身を修正した日時) において、POSIX で規定されているのは`-mtime`だけであり、しかも後ろには単純な数字しか指定できない。つまり現在から 1 日（=86400 秒）単位での新旧比較しかできない。その代わり`-newer`というオプションが用意されており、これを使うとそのファイルより新しいかどうかという条件指定ができるため、これで辛うじて新旧比較ができるようになる。タイムスタンプはどの環境でも秒単位まであるから、つまり秒単位まで新旧比較ができることになる。

ただ、`-newer`というオプション自体は「より新しい（等しいものはダメ）」という判定のみであるので、色々と工夫が必要である。「より古い」を判定したければ否定演算子を併用して「目的の日時の 1 秒前のより新しくない」とすることになるし、「等しい」にしたければ「目的の日時の 1 秒前のより新しく、かつ、目的の日時より新しくない」というように 1 秒ずらして前後から挟み込む。

現在日時からの相対で指定したい場合は、前に紹介したレシピを活用して時間の計算をして絶対日時を求め、同様の比較をすればよいというわけだ。

参照

→レシピ 3.3（シェルスクリプトで時間計算を一人前にこなす）

レシピ 3.5　CSV ファイルを読み込む

問題

Excel からエクスポートした CSV ファイルの任意の行の任意の列を読み出したい。しかし実際読み出すとなったら、列区切りとしてカンマと値としてのカンマを区別しなければいけなかったり、行区切りとしての改行と値としての改行を区別しなければいけなかったり、さらに値としてのカンマや改行を区別するためのダブルクォーテーション記号を意識しなければいけなかったり、大変だ。

回答

sed や AWK を駆使すれば POSIX の範囲でパーサー（解析プログラム）の作成が可能である。原理の解説は後回しにするが、そうやって制作した CSV パーサー “parsrc.sh” があるので、それをダウンロード[*3]して用いる。

例えば、次のような CSV ファイル（sample.csv）があったとする。

```
aaa,"b""bb","c
cc",d d
"f,f"
```

これを次のようにして parsrc.sh に掛けると、第 1 列:元の値のあった行番号、第 2 列:元の値のあった列番号、第 3 列:値、という 3 つの列から構成されるテキストデータに変換される。

```
$ ./parsrc.sh sample.csv ↵
1 1 aaa
1 2 b"bb
1 3 c¥ncc   ←値としての改行は “¥n” に変換される（オプションで変更可能）
1 4 d d
2 1 f,f
$
```

よって、後ろにパイプ越しにコマンドを繋げば「任意の行の任意の列の値を取得」もできるし、「全ての行について指定した列の値を抽出」などということもできる。

[*3] https://github.com/ShellShoccar-jpn/Parsrs/blob/master/parsrc.sh にアクセスし、そこにあるソースコードをコピー&ペーストしてもよいし、あるいは “RAW” と書かれているリンク先を「名前を付けて保存」してもよい。

(a) 1 行目の 3 列目の値を取得

```
$ ./parsrc.sh sample.csv | grep '^1 3 '| sed 's/^[^ ]* [^ ]* //' ↵
c¥ncc
$
```

(b) 全ての行の 1 列目を抽出

```
$ ./parsrc.sh sample.csv | grep -E '^[^ ]+ 3 '| sed 's/^[^ ]* [^ ]* //' ↵
aaa
f,f
$
```

解説

CSV ファイルは、Microsoft Excel とデータのやり取りをするには便利なフォーマットだが、AWK などの標準 UNIX コマンドで扱うにはかなり面倒だ。しかしできないわけではない。先程利用した parsrc.sh というプログラムのソースコードは長すぎて載せられないが、どのようにしてデータの正規化[*4]をしたのかについて、概要を説明することにする。

CSV ファイル（RFC 4180）の仕様を知る

その前に、加工対象となる CSV ファイルのフォーマット仕様について知っておく必要がある。CSV ファイルの構造にはいくつかの方言があるのだが、最も一般的なものが RFC 4180 で規定されている。Excel で扱える CSV もこの形式に準拠したものだ。主な特徴は次のとおりである。

1. 列はカンマ “,” で区切り、行は改行文字で区切る。
2. 値としてこれらの文字が含まれる場合は、その列全体をダブルクォーテーション “"” で囲む。
3. 値としてダブルクォーテーションが含まれる場合は、その列全体をダブルクォーテーション “"” で囲んだうえで、値としてのダブルクォーテーション文字については 1 つにつき 2 つのダブルクォーテーションの連続""で表現する。

この 3 番目の仕様が、CSV パーサーを作る上で重要である。この仕様があるおかげで、値としての改行を見つけだすことができる。最初に示した CSV ファイルの例をもう一度見てみよう。

```
aaa,"b""bb","c   ← ダブルクォーテーションが奇数個
cc",d d          ← ダブルクォーテーションが奇数個
"f,f"            ← ダブルクォーテーションが偶数個
```

最初の 2 行はダブルクォーテーションの数がそれぞれ奇数個である。これは列を囲っているダブルクォーテーションがその行単独では閉じてないということを意味している。つまり本来は同一行なのだが、値としての改行が含まれているために分割されてしまっているのだ。値としてのダブルクォーテーションは元々の 1 つを 2 個で表現しているから、偶数奇数の判断に影響を及ぼさない。よって、**ダブルクォーテーションが奇数個の行が出現したら、次に奇数個の行が出現するまで、同一行と判断することができる。**

[*4] 都合の良い形式に変換すること。この場合、UNIX コマンドで扱い易いように「行番号、列番号、値」という並びに変換した作業を指す。

仕様に基づき、実装する

■1) 値としての改行の処理　この性質がわかれば正規化作業も見通しがつく。AWK で 1 行ずつダブルクォーテーション文字数を数え、奇数個の行が出てきたら、次にまた奇数個の行が出てくるまで行を結合していけばいい。ただし、単純に結合するとそこに値としての改行文字があったことがわからなくなってしまうので通常のテキストファイルには用いられないコントロールコード（<0x0F> を選んだ）を挿んだうえで結合していく。

■2) 値としてのダブルクォーテーションの処理　次は、値としてのカンマに反応しないように気をつけながら、行の中に含まれる各列を単独の行へと分解していく。基本的には行の中にカンマが出現するたびに改行に置換していけばいいのだが、2 つの点に気をつけなければならない。1 つは値としてのカンマを無視することであるが、これは先にダブルクォーテーションが出現していた場合は次のダブルクォーテーションが出現するまでに存在するカンマを無視するように正規表現置換をすればいい。ただ、値としてのダブルクォーテーションがあると失敗してしまうので、実は手順 1) の前でそれ（ダブルクォーテーション文字の 2 連続）を別のコントロールコード（<0x0E> とした）にエスケープしておくのだ。そしてもう一つ気をつけなければならないのが、元々の改行と列区切りカンマを置換して作った改行を区別できるようにしなければならないということだ。その目的で、元々の改行が出現した時点で更に第 3 のコントロールコード（レコード区切りを意味する <0x1E> を選んだ）を挿むようにした。

■3) 列と行の数を数えて番号を付ける　あとは、改行の数を数えれば作業は大方終了だ。改行が来るたびに列番号を 1 増やしてやればよいが、元々の改行の印として付けた第 3 のコントロールコードが来た時点で 1 に戻してやる。最後は、そうやって出力したコードに残っている第 2 のコントロールコードを戻す。これは何だったかというと値としてのダブルクォーテーションであった。最初、変換した時点では 2 個のダブルクォーテーションで表現されていたが、元々は 1 個のダブルクォーテーションを意味していたのだから 1 個に戻してやればよい。

概要は掴めただろうか。掴めなかったとしても、とにかく POSIX の範囲のコマンドだけでできるんだということがわかれば十分だ。

参照

→レシピ 3.6（JSON ファイルを読み込む）
→レシピ 3.7（XML、HTML ファイルを読み込む）

レシピ 3.6　JSON ファイルを読み込む

問題

WebAPI を叩いて得られた JSON ファイルの任意の箇所の値を読み出したい。しかし、先程解説された CSV ファイルの比ではなく構造が複雑だ。それでもできるのか？

回答

JSON であっても sed や AWK を駆使すれば、やはり POSIX の範囲でパーサーが作れる。解説は後回しにして、制作した JSON パーサー “parsrj.sh” をダウンロード[*5]して使ってもらいたい。

例えば、次のような CSV ファイル（sample.json）があったとする。

```
{"会員名" : "文具 太郎",
 "購入品" : [ "はさみ",
              "ノート (A4, 無地)",
              "シャープペンシル",
              {"取寄商品" : "替え芯"},
              "クリアファイル",
              {"取寄商品" : "6 穴パンチ"}
            ]
}
```

これを次のようにして parsrj.sh に掛けると、第 1 列:元の値のあった場所（JSONPath 形式[*6]）、第 2 列:値、という 2 つの列から構成されるテキストデータに変換される。

```
$ ./parsrj.sh sample.json ↵
$.会員名 文具 太郎
$.購入品 [0] はさみ
$.購入品 [1] ノート (A4, 無地)
$.購入品 [2] シャープペンシル
$.購入品 [3].取寄商品 替え芯
$.購入品 [4] クリアファイル
$.購入品 [5].取寄商品 6 穴パンチ
$
```

よって、後ろにパイプ越しにコマンドを繋げば「任意の場所の値を取得」ができる。例えば次のような具合だ。

(a) 購入品の 2 番目（番号 1）を取得する

```
$ ./parsrj.sh sample.json | grep '^¥$¥.購入品¥[1¥]'| sed 's/^[^ ]* //' ↵
ノート (A4, 無地)
$
```

(b) 全ての取寄商品名を抽出

```
$ ./parsrj.sh sample.json | awk '$1~/取寄商品$/'| sed 's/^[^ ]* //' ↵
替え芯
6 穴パンチ
$
```

[*5] https://github.com/ShellShoccar-jpn/Parsrs/blob/master/parsrj.sh にアクセスし、そこにあるソースコードをコピー&ペーストしてもよいし、あるいは “RAW” と書かれているリンク先を「名前を付けて保存」してもよい。

[*6] JSON データは階層構造になっているので、同様に階層構造をとるファイルパスのようにして一行で書き表せる。その記法が JSONPath である。詳細は http://goessner.net/articles/JsonPath/参照。

JSON にエスケープ文字が混ざっている場合

JSON は、コントロールコードや時にはマルチバイト文字をエスケープして格納している事がある。例えば “¥n” や “¥t” はそれぞれタブと改行文字であるし、“¥uXXXX”（XXXX は 4 桁の 16 進数）は Unicode 文字である。このような文字を元に戻すフィルターコマンド “unescj.sh” も同じ場所にリリースした[*7]。

詳細な説明は parsrj.sh と unescj.sh のソースコード冒頭に記したコメントを見てもらうことにして割愛するが、parsrj.sh で解読したテキストファイルをパイプ越しに unescj.sh に与えれば解読してくれる。もちろん unescj.sh も POSIX の範囲で書かれている。

解説

CSV がパースできたのと同様に、JSON のパースも POSIX の範囲でできる。本章の冒頭で述べたことだが、POSIX に含まれる sed や AWK はチューリングマシンの要件を満たしているのだから、それらを使えば理論的にも可能なのだ。CSV の時と同様、JSON にも値の位置を示すための記号と、同じ文字ではあるものの純粋な値としての記号があるが、冷静に手順を考えればきちんと区別・解読することができるのだ。とは言うものの具体的にどのようにして実現したのかを解説するのは大変なので、どうしても知りたい人はソースコード読んでもらうことにして、ここでの説明は割愛する。

しかし、JSON パーサーとしては “jq” という有名なコマンドが既に存在する。にもかかわらず、筆者はなぜ再発明したのか。確かに、こちらのコマンドなら POSIX 原理主義に基づいているため、「どこでも動く」「コンパイルせずに動く」「10 年後も 20 年後も、たぶん動く」の三拍子が揃っているという利点もあるのだが、真の理由はまた別のところにあったので、ここで語っておきたい。この後のレシピで XML パーサーを作った話を述べるが、同様の理由であるのでまとめて語ることにする。

JSON & XML パーサーという「車輪の再発明」の理由

jq や xmllint 等、UNIX 哲学に染まりきっていない

シェルスクリプトで、JSON を処理したいといったら jq コマンド、XML を処理したいといったら xmllint や hxselect（html-xml-utils というユーティリティーの 1 コマンド）、あるいは MacOS X の xpath といったコマンドを思い浮かべるかもしれない。そして、それらは「便利だ」という声をちらほら聞くのだが、私はちっとも便利に思えない。試しに使ってみても「なぜこれで満足できる？」とさえ思う。理由はこうだ。

■理由 1. 一つのことをうまくやっていない

UNIX 哲学の一つとしてよく引用されるマイク・ガンカーズの教義に

1. 小さいものは美しい。
2. 1 つのプログラムには 1 つのことをうまくやらせよ。

というものがある。しかし、まずこれができていない。jq や xmllint 等は、データの正規化 (都合の良い形式に変換する) 機能とデータの欲しい部分だけを抽出する部分抽出機能を分けていない。むしろ前者をすっ飛ばして後者だけやっているように思う。

でも UNIX 使いとしては、**部分抽出といったら grep や AWK** を使い慣れているわけで、それらでできるようにしてもらいたいと思う。部分抽出をするために、**jq や xmllint 等独自の文法をわざわざ覚えたくない**し。

[*7] https://github.com/ShellShoccar-jpn/Parsrs/blob/master/unescj.sh

だから、正規化だけをやるようなコマンドであってほしかった。

■理由 2. フィルターとして振る舞うようになりきれてない
同じく教義の一つに

9. 全てのプログラムはフィルターとして振る舞うように作れ。

というものがある。フィルターとは、入力されたものに何らかの加工を施して出力するものをいうが、jq や xmllint 等は、出力の部分に注目するとフィルターと呼ぶには心もとない気がする。

なぜなら、これらのプログラムはどれも **JSON や XML 形式のまま出力**される。しかし、他の標準 UNIX コマンドからは扱いづらい。AWK,sed,grep,sort,head,tail,……などなど、**標準 UNIX コマンドの多くは、行単位あるいは列単位 (空白区切り) のデータを加工するのに向いた仕様**になっているため、JSON や XML 形式のままだと結局扱いづらい。

だから、行や列の形に正規化するコマンドであってほしかった。

無いものは作る。シェルスクリプトとパイプを駆使して。

そうして作ったパーサーが、このレシピやそしてこの後のレシピ 3.7 で紹介したものだ。GitHub に置いてあるソースコード[*8]を見れば明白だが、これらのパーサーもまた、シェルスクリプトを用い、AWK、sed、grep、tr 等をパイプで繋ぐだけで実装した。

これまた UNIX 哲学の教義だが、

6. ソフトウェアは「てこ」。最小の労力で最大の効果を得よ。
7. 効率と移植性を高めるため、シェルスクリプトを活用せよ。

というのがある。その教義に従って実際に作ってみると、シェルスクリプトや UNIX コマンド、パイプというものがいかに偉大な発明であるか思い知らされた。

無ければ自分で作る。由緒正しい UNIX の教本にも、「置いてあるコマンドは見本みたいなものだから、無いものは自分達で作りなさい」と書かれているそうだ。それに、自分で作れば、対象概念の理解促進にもつながる。

参照

→レシピ 3.5（CSV ファイルを読み込む）
→レシピ 3.7（XML、HTML ファイルを読み込む）

レシピ 3.7　XML、HTML ファイルを読み込む

問題

WebAPI を叩いて得られた XML ファイルの任意の箇所の値を読み出したい。CSV、JSON と来たらやっぱり XML もできるんでしょ？
もしそれができたら、HTML のスクレイピングもできるようになるのだろうか。

[*8] https://github.com/ShellShoccar-jpn/Parsrs

回答

XML ももちろん可能だ。sed や AWK を駆使すれば、やはり POSIX の範囲でパーサーが作れる。ただ、HTML のスクレイピングに流用できることはあまり期待しない方がいい。理由は、HTML は閉じタグが無いなど文法が間違っていても Web ブラウザーが許容するおかげでそのままになっているものがあるし、更には
 など、閉じタグが無いことが文法的にも認められているものがあり、そういった XML 的に文法違反のものには通用しないからだ。

さて解説は後回しにして、制作した XML パーサー “parsrx.sh” をダウンロード[*9]して使ってもらいたい。

例えば、次のような XML ファイル（sample.xml）があったとする。

```
<文具購入リスト 会員名="文具 太郎">
  <購入品>はさみ</購入品>
  <購入品>ノート（A4，無地）</購入品>
  <購入品>シャープペンシル</購入品>
  <購入品><取寄商品>替え芯</取寄商品></購入品>
  <購入品>クリアファイル</購入品>
  <購入品><取寄商品>６穴パンチ</取寄商品></購入品>
</文具購入リスト>
```

これを次のようにして parsrx.sh に掛けると、第 1 列:元の値のあった場所（XPath 形式[*10]）、第 2 列:値、という 2 つの列から構成されるテキストデータに変換される。

```
$ ./parsrx.sh sample.xml ↵
/文具購入リスト/@会員名 文具 太郎
/文具購入リスト/購入品 はさみ
/文具購入リスト/購入品 ノート（A4，無地）
/文具購入リスト/購入品 シャープペンシル
/文具購入リスト/購入品/取寄商品 替え芯
/文具購入リスト/購入品
/文具購入リスト/購入品 クリアファイル
/文具購入リスト/購入品/取寄商品 ６穴パンチ
/文具購入リスト/購入品
/文具購入リスト ¥n ¥n ¥n ¥n ¥n ¥n ¥n
$
```

よって、後ろにパイプ越しにコマンドを繋げば「任意の場所の値を取得」ができる。例えば次のような具合だ。

[*9] https://github.com/ShellShoccar-jpn/Parsrs/blob/master/parsrx.sh にアクセスし、そこにあるソースコードをコピー&ペーストしてもよいし、あるいは “RAW” と書かれているリンク先を「名前を付けて保存」してもよい。

[*10] XML データは階層構造になっているので、同様に階層構造をとるファイルパスのようにして一行で書き表せる。その記法が XPath である。詳細は http://www.w3.org/TR/xpath-31/ 参照。

(a) 購入品の 2 番目を取得する

```
$ ./parsrx.sh sample.xml | awk '$1~/取寄商品$/'| sed '2s/^[^ ]* //' ↵
ノート (A4, 無地)
$
```

(b) 全ての取寄商品名を抽出

```
$ ./parsrx.sh sample.xml | awk '$1~/取寄商品$/'| sed 's/^[^ ]* //' ↵
替え芯
6 穴パンチ
$
```

解説

POSIX 範囲内で実装した CSV、JSON パーサーを作ったのなら当然次は XML パーサーであるが、もちろん作れた。ただ、XML はプロパティーとしての値とタグで囲まれた文字列としての値というように値が 2 種類あったり、コメントが許されていたりするため、さらに複雑であった。複雑であるため、こちらもどうやって実現したのかをどうしても知りたい人はソースコード読んでもらうことにして、説明は割愛する。

HTML テキストへの流用

最初でも触れたが HTML テキストのパーサーとして使えるかどうかは場合による。厳密書かれた XHTML になら使えるが、既に述べたように多くの Web ブラウザーはいい加減な HTML を許容するうえ、HTML の規格自体が閉じタグ無しを許したりするのでそのような HTML テキストが与えられると、うまく動かないだろう。

ただし、「いい加減な記述が混ざっている HTML ではあるが、中にある正しく書かれた <table> の中身だけ取り出したい」という場合、sed コマンド等を使って予めその区間だけ切り出しておけば流用可能である。

参照

→レシピ 3.5（CSV ファイルを読み込む）
→レシピ 3.6（JSON ファイルを読み込む）

レシピ 3.8　全角・半角文字の相互変換

問題

大文字・小文字を区別せず、更に全角・半角も区別せずにテキスト検索がやりたい。全角文字を半角に変換することさえできればあとは簡単なのだが。

回答

下記のような正直な処理を行うプログラムを書けばよい。

- テキストデータを 1 バイトずつ読み、各文字が何バイト使っているのかを認識しながら読み進めていく。

- その際、半角文字に変換可能な文字に遭遇した場合は置換する。

全角文字→半角文字変換コマンド “han”

とは言っても毎回それを書くのも大変だ。しかし例によって POSIX の範囲で実装し、コマンド化したものが GitHub に公開されている。“han” という名のコマンドだ。これは USP 研究所が Open usp Tukubai というシェルスクリプト開発者向けコマンドセット[*11]として公開しているものの派生物であり、オリジナルが Python で実装されているところを、321516 氏が POSIX 原理主義に基づいて書き直した完全互換コマンドである。これをダウンロード[*12]して用いる。

例えば、次のようなテキストファイル（enquete.txt）があったとする。

```
#name          ans1  ans2
Ｍｏｇａｍｉ  yes   no
Kaga           no    yes
fubuki         yes   yes
ｍｕｔｓｕ    no    no
Ｓｈｉｍａｋａｚｅ  no    yes
```

アンケート回答がまとまっているのだが、回答者によって自分の名前を全角で打ち込んだり半角で打ち込んだり、まちまちというわけだ。回答者名で検索したいとなった時、検索する側はいちいち大文字・小文字や全角・半角を区別したくない。このよう時に、han コマンドを使うのである。

次のようにして han コマンドに掛けた後、tr コマンドで大文字を全て小文字に変換する（その後で AWK に掛けているのは見やすさのためだ）。

```
$ ./han enquete.txt | tr A-Z a-z | awk '{printf("%-10s %-4s %-4s¥n",$1,$2,$3);}'↵
#name      ans1 ans2
mogami     yes  no
kaga       no   yes
fubuki     yes  yes
mutsu      no   no
shimakaze  no   yes
$
```

こうしておけば、半角英数字で簡単に回答者名の grep 検索が可能だ。

尚、この **han コマンドは UTF-8 のテキストにしか対応しておらず**、JIS や Shitf JIS、EUC-JP テキストには対応していない。そのような文字を扱いたい場合は、iconv コマンドを用いたり、nkf コマンド（POSIX ではないが）を用いて予め UTF-8 に変換しておくこと。

半角文字→全角文字変換コマンド “zen”

今回の問題では必要なかったが、han コマンドとは逆に、文字を全角に変換するコマンドもある。“zen” という名のコマンドだ。これも USP 研究所が Open usp Tukubai として公開しているコマンドセットが元になっていて、321516 氏が同様に POSIX 原理主義に基づいて書き直したものが公開されている。これをダウンロー

[*11] https://uec.usp-lab.com/TUKUBAI/CGI/TUKUBAI.CGI?POMPA=ABOUT

[*12] https://github.com/321516/Open-usp-Tukubai/blob/master/COMMANDS.SH/han にアクセスし、そこにあるソースコードをコピー&ペーストしてもよいし、あるいは “RAW” と書かれているリンク先を「名前を付けて保存」してもよい。

ド[*13]して用いる。

通常は全ての文字を全角文字に変換するのだが、-k オプションを付けると半角カタカナのみ全角変換することができる。

```
$ echo 'ﾊﾝｶｸ文字は e-mail では使えません。'| ./zen -k↵
ハンカク文字は e-mail では使えません。
$
```

これは e-mail 送信用のテキストファイルを作る際に有用だ。

尚、zen コマンドもやはり UTF-8 専用である。

解説

全角混じりのテキストだって取り扱いを諦めることはない。一文字一文字愚直に、変換可能なものを変換していけばいいだけだ。ただ、その際に問題になるのはマルチバイトの扱いである。1 バイトずつ読んだ場合、それがマルチバイト文字の終端なのか、それとも途中なのかということを常に判断しなければならない。

han、zen コマンドは文字エンコードが UTF-8 前提で作られているが、そのためには UTF-8 の各文字のバイト長を正しく認識しなければならない。その情報は、Wikipedia 日本語版の UTF-8 のページにも記載されている。1 文字読み込んでみてそのキャラクターコードがどの範囲にあるかということを判定していくと、1 バイトから 6 バイトの範囲で長さを決定することができる。han、zen にはそのようなルーチンが実装されている。

そして 1 文字分読み取った結果、それと対になる半角文字あるいは全角文字が存在する時は、元の文字ではなく用意していた対の文字を出力すればよい。この時も POSIX 版 AWK がやっていることはとても単純だ。対になる文字を全て AWK の連想配列に登録しておき、要素が存在すれば代わりに出力しているに過ぎない。ただし、半角カタカナから全角カタカナへの変換の時には注意事項がある。それは濁点、半濁点の処理だ。例えば、半角の「ﾊ」の直後に半角の「ﾞ」が連なっていたら「ﾊﾞ」ではなくて「パ」に変換しなければならないので、「ﾊ」が来た時点ですぐに置換処理をしてはならずに次の文字を見てからにしなければならないのだ。

このようにやり方を聞いて「なんてベタな書き方だ」と笑うかもしれない。しかし**速度の問題が生じない限り、プログラムはベタに書く方がいい**。その方が、他人にとっても、そして将来の自分にとっても、メンテナンスしやすいプログラムになる。UNIX 哲学の教義その 1

> Small is beautiful.

のとおりだ。

参照

→レシピ 3.9（ひらがな・カタカナの相互変換）

[*13] https://github.com/321516/Open-usp-Tukubai/blob/master/COMMANDS.SH/zen にアクセスし、そこにあるソースコードをコピー&ペーストしてもよいし、あるいは “RAW” と書かれているリンク先を「名前を付けて保存」してもよい。

レシピ 3.9　ひらがな・カタカナの相互変換

問題

名簿入力フォームで名前とふりがなが集まったのだが、ふりがなが人によってひらがなだったりカタカナだったりするので統一したい。どうすればいいか。

回答

全角・半角の相互変換と方針は似ていて、基本方針は

- テキストデータを 1 バイトずつ読み、各文字が何バイト使っているのかを認識しながら読み進めていく。
- その際、対になるひらがなあるいはカタカナに変換可能な文字に遭遇した場合は置換する。

である。尚、ひらがなは全角文字にしか存在しない[*14]ため、全角文字前提での話とする。半角カタカナを扱いたい場合は、レシピ 3.8（全角・半角文字の相互変換）によって全角に直してから参照すること。

ひらがな→カタカナ変換コマンド “hira2kata”

例によって POSIX の範囲で実装し、コマンド化したものが GitHub に公開されている。“hira2kata” という名のコマンドだ。これは 321516 氏が、USP 研究所の Open usp Tukubai というシェルスクリプト開発者向けコマンドセット[*15]にある han、zen コマンドのインターフェースに似せる形で作ったものである。もちろん POSIX の範囲でだ。これをダウンロード[*16]して用いる。

例えば、次のようなテキストファイル（furigana.txt）があったとする。

```
#No. フリガナ
い   もがみ
ろ   カガ
は   ふぶき
に   ムツ
ほ   ぜかまし
```

問題文にもあったように、回答者によってふりがなをひらがなで入力したりカタカナで入力したりまちまちになっている。回答者名で検索したいとなった時、検索する側はいちいちひらがなかカタカナかを区別したくない。このよう時に、次のようにして hira2kata を使うのである。

[*14] MSX など半角ひらがなを持っているコンピューターはあるのだけど一般的ではない。

[*15] https://uec.usp-lab.com/TUKUBAI/CGI/TUKUBAI.CGI?POMPA=ABOUT

[*16] https://github.com/ShellShoccar-jpn/misc-tools/blob/master/hira2kata にアクセスし、そこにあるソースコードをコピー&ペーストしてもよいし、あるいは “RAW” と書かれているリンク先を「名前を付けて保存」してもよい。

```
$ ./hira2kata 2 furigana.txt ↵
#No. フリガナ
い　モガミ
ろ　カガ
は　フブキ
に　ムツ
ほ　ゼカマシ
$
```

こうしておけば、全角カタカナで簡単に回答者名の grep 検索が可能になるし、50 音順ソートもできるようになる。注意すべきは、この例では hira2kata コマンドの第 1 引数に “2” と書いてあるところである。これは、第 2 フィールドだけ変換せよという意味である。よって第 1 フィールドの数字はそのままになっている。仮に “2” という引数無しにファイル名だけ指定すると、フィールドという概念無しに、テキスト中にある全てのひらがなを変換しようとする。よって、その場合第 1 フィールドの「いろは……」もカタカナになる。

尚、この **hira2kata コマンドは UTF-8 のテキストにしか対応しておらず**、JIS や Shitf JIS、EUC-JP テキストには対応していない。そのような文字を扱いたい場合は、iconv コマンドを用いたり、nkf コマンド（POSIX ではないが）を用いて予め UTF-8 に変換しておくこと。

それから、このサンプルデータで「ほ」行の島風さんだけ右から書いてあるが、**こういうのはどーしよーもない！**昭和じゃないんだから左から書くように言っておいってもらいたい。

カタカナ→ひらがな変換コマンド “kata2hira”

上の例ではカタカナに統一したが、逆にひらがなに統一してもよい。その場合は “kata2hira” という名のコマンドを使う。これも、321516 氏が USP 研究所の Open usp Tukubai というシェルスクリプト開発者向けコマンドセット[*17]にある han、zen コマンドのインターフェースに似せる形で、POSIX 範囲内で作ったものである。あわせてダウンロード[*18]しておくとよいだろう。

解説

前のレシピで全角文字を半角文字に変換するコマンドが作れたのだから、半角文字に変換する代わりにひらがな・カタカナの変換をするのも大したことはない。

マルチバイト文字なので、半角・全角変換と同様に、1 バイトずつ読んだ場合、それがマルチバイト文字の終端なのか、それとも途中なのかということを常に判断しなければならないのだが、その後の置換作業で一工夫してある。

半角全角変換の際は完全に連想配列に依存していたが、ひらがな・カタカナ変換においては、高速にするために使用を控えている。その代わりにキャラクターコードを数百番ずつシフトするような計算を行っている。UTF-8 においては、ひらがなとカタカナはユニコード番号が数十バイト離れたところにそれぞれマッピングされている[*19]ので、それを見ながらユニコード番号を数百番ずらして目的の文字を作っているというわけだ。

[*17] https://uec.usp-lab.com/TUKUBAI/CGI/TUKUBAI.CGI?POMPA=ABOUT

[*18] https://github.com/ShellShoccar-jpn/misc-tools/blob/master/kata2hira にアクセスし、そこにあるソースコードをコピー&ペーストしてもよいし、あるいは “RAW” と書かれているリンク先を「名前を付けて保存」してもよい。

[*19] 『オレンジ工房』さんの UTF-8 の文字コード表 全角ひらがな・カタカナというページ参照
→ http://orange-factory.com/sample/utf8/code3-e3.html

参照

→レシピ 3.8（全角・半角の相互変換）

第 4 章

POSIX 原理主義テクニック – Web 編

前章の POSIX 原理主義テクニックは堪能してもらえただろうか。筆者の周囲では JSON、XML のパースができることに驚いてくれる人が多いが、**驚くのはまだ早い！** Web アプリケーションを作るうえで役立つ数々のレシピはこれから紹介するのだから。

そもそも POSIX の範囲で Web アプリケーションを作ること自体、驚く以前に、信じられないようだ。しかし、本章のレシピを読めば現実味が湧く事だろう。

事実、ショッピングカートの「**シェルショッカー**」、東京メトロのオープンデータに基づく列車接近情報表示アプリケーション「**メトロパイパー**」は、これらのレシピを活用して作ったものだ。信じられないなら、これらのキーワードで Web 検索して動作画面やソースコードを見てみるといい。

レシピ 4.1　URL デコードする

問題

Web サーバーのログを見ていると、検索ページからジャンプしてきている形跡があった。しかし、検索キーワードは URL エンコードされた状態であり、デコードしないとわからないのでデコードしたい。

回答

そんなに難しい仕事でないから素直に書いて作る。基本的には正規表現で “`%[0-9A-Fa-f]{2}`” を検索し、見つかるたびに printf 関数を使ってその 16 進数に対応するキャラクターに置き換えればよい。AWK で書くならこんな感じだ。

■URL デコードするコード

```
env i- awk '
BEGIN {
  # --- prepare
  LF  = sprintf("¥n");
  OFS = "";
  ORS = "";
  # --- prepare decoding
  for (i=0; i<256; i++) {
    l  = sprintf("%c",i);
    k1 = sprintf("%02x",i);
    k2 = substr(k1,1,1) toupper(substr(k1,2,1));
    k3 = toupper(substr(k1,1,1)) substr(k1,2,1);
    k4 = toupper(k1);
    p2c[k1]=l;p2c[k2]=l;p2c[k3]=l;p2c[k4]=l;
  }
  # --- decode
  while (getline line) {
    gsub(/¥+/, " ", line);
    while (length(line)) {
      if (match(line,/%[0-9A-Fa-f][0-9A-Fa-f]/)) {
        print substr(line,1,RSTART-1), p2c[substr(line,RSTART+1,2)];
        line = substr(line,RSTART+RLENGTH);
      } else {
        print line;
        break;
      }
    }
    print LF;
  }
}'
```

1 文字ではなく 1 バイトずつ処理する必要があるので “env -i” を AWK の手前に付けて、ロケール環境変数の影響を受けないようにする。

解説

文字を 1 バイト毎に、16 進数 2 桁表現でアスキーコード化し、その先頭に “%” 文字を付けるエンコード方式を**パーセントエンコーディング**と呼ぶ。ただし URL に用いる文字のうち特殊な意味を持つものだけをパーセントエンコーディングするとともに半角スペースは “%20” ではなく “+” にエンコードする場合を、「URL エンコーディング」とか「URL エンコード」などと呼んだりする。これは、RFC 3986 の Section2.1 で定義されている。

このエンコーディングのルールさえ理解できれば、デコーダーを作ることなど大したことではない。例えば Web 検索すると urlendec というパッケージ[*1]が見つかる。しかし POSIX 原理主義者たるものそういったものに安易に頼ってはいけない。なんとこのツールは 2014/12 現在 32bit 専用らしい。もしこのソフトを愛用している人が 32bit 環境から 64bit 環境に移行しようとしたら痛い目を見る。（かつての筆者）

[*1] http://www.whizkidtech.redprince.net/urlendec/

参照

→ RFC 3986 文書[*2]
→レシピ 4.2（URL エンコードする）

レシピ 4.2　URL エンコードする

問題

WebAPI を叩きたいのだが、パラメーターには URL エンコーディングされた文字列を渡さなければならない。どうすればいいか？

回答

URL デコードよりも多少面倒だが、やはりそんなに難しい仕事でないから素直に書いて作る。基本的には文字列を 1 バイトずつ読み込んで、2 桁 16 進数（大文字）のアスキーコードにしながら先頭に “%” を付ける。「多少面倒」というのは、変換の必要がある文字かどうかを判断して必要な場合のみ変換するということだ。

その注意点を踏まえながら AWK で書くならこんな感じだ。

■URL エンコードするコード

```
env -i awk '
BEGIN {
  # --- prepare
  LF = sprintf("¥n");
  OFS = "";
  ORS = "";
  # --- prepare encoding
  for(i= 0;i<256;i++){c2p[sprintf("%c",i)]=sprintf("%%%02X",i);}
  c2p[" "]="+";
  for(i=48;i< 58;i++){c2p[sprintf("%c",i)]=sprintf("%c",i);    }
  for(i=65;i< 91;i++){c2p[sprintf("%c",i)]=sprintf("%c",i);    }
  for(i=97;i<123;i++){c2p[sprintf("%c",i)]=sprintf("%c",i);    }
  c2p["-"]="-"; c2p["."]="."; c2p["_"]="_"; c2p["~"]="~";
  # --- encode
  while (getline line) {
    for (i=1; i<=length(line); i++) {
      print c2p[substr(line,i,1)];
    }
    print LF;
  }
}'
```

1 文字ではなく 1 バイトずつ処理する必要があるので “env -i” を AWK の手前に付けて、ロケール環境変数の影響を受けないようにするのはデコードの時と同じである。

[*2] http://www.ietf.org/rfc/rfc3986.txt

解説

URL エンコーディングとは何かについてはレシピ 4.1（URL デコードする）で説明したので省略する。そこで不足していた説明としては、エンコーディングする必要のある文字が何かということだが、逆に必要の無い文字は次のとおりである。

> アルファベット（A〜Z、a〜z）、数字（0〜9）、ハイフン（-）、ピリオド（.）、アンダースコア（_）、チルダ（~）

これらの文字がについては、エンコーディングせずにそのまま出力するのだが 1 つ 1 つ判定するのは大変であるので、「回答」で示したコードのように AWK の連想配列を使うのが良いだろう。

おススメはしないけど

GNU 版 sed を使うと

```
s="ここに URL エンコードした文字"
echo -e $(echo -n "$s" | od -An -tx1 -v -w99999 | tr ' ' % | sed 's/%20/+/g' | sed 's/%¥(2[de]¥|
3[0-9]¥|4[^0]¥|5[0-9AaFf]¥|6[^0]¥|7[0-9]¥)/¥¥x¥1/g')
```

というワンライナーにできるらしいが、POSIX 原理主義者ならとーぜん禁じ手だ。

参照

→レシピ 4.1（URL デコードする）
→ RFC 3986 文書[*3]

レシピ 4.3　CGI 変数の取得（GET メソッド編）

問題

Web ブラウザーから送られてくる CGI 変数を読み取りたい。
ただし GET メソッド（環境変数 “REQUEST_METHOD” が “GET” の場合）である。

回答

CGI 変数を読み出すのに便利な 2 つのコマンド（“cgi-name” 及び “namread”）が Open usp Tukubai で提供されており、これらを POSIX の範囲で書き直したものが 321516 氏によって提供されているので、まずこれらをダウンロード[*4]する。

今、Web ブラウザーが次のような HTML に基づいて CGI 変数を送ってくるものとしよう。

[*3] http://www.ietf.org/rfc/rfc3986.txt
[*4] https://github.com/321516/Open-usp-Tukubai/blob/master/COMMANDS.SH/cgi-name、及び https://github.com/321516/Open-usp-Tukubai/blob/master/COMMANDS.SH/nameread にアクセスし、そこにあるソースコードをコピー&ペーストしてもよいし、あるいは “RAW” と書かれているリンク先を「名前を付けて保存」してもよい。

■Web ブラウザーが送信する元になる HTML

```
<form action="form.cgi" method="GET">
  <dl>
    <dt>名前</dt>
      <dd><input type="text" name="fullname" value="" /></dd>
    <dt>メールアドレス</dt>
      <dd><input type="text" name="email" value="" /></dd>
  </dl>
  <input type="submit" name="post" value="送信" />
</form>
```

GET メソッド（環境変数 “REQUEST_METHOD” が “GET”）の場合、CGI 変数は環境変数 “QUERY_STRING” の中に入っているので、まず cgi-name を使ってこれを正規化して一時ファイルに格納する。あとは nameread コマンドを使い、取り出したい変数をシェル変数等に取り出せばよい。

まとめると次のようになる。

■前述のフォームから送られてきた CGI 変数を受け取るシェルスクリプト（form.cgi）

```
#! /bin/sh

Tmp=/tmp/${0##*/}.$$  # 一時ファイルの元となる名称

printf '%s' "${QUERY_STRING:-}" |
cgi-name                        > $Tmp-cgivars  # 正規化し、一時ファイルに格納

fullname=$(nameread fullname $tmp-cgivars)      # CGI 変数"fullname"を取り出す
email=$(nameread email $tmp-cgivars)            # CGI 変数"email"を取り出す

(ここで何らかの処理)

rm -f $Tmp-*                                    # 用が済んだら一時ファイルを削除
```

尚、元のデータには漢字や記号が含まれていて、URL エンコードされていてももちろん構わない。それらのデコードは cgi-name コマンドが済ませてくれているのだ。

解説

Web ブラウザーから情報を受け取りたい場合によく用いられるのが CGI 変数であるが、その送られ方にはいくつかの種類がある。「回答」で述べたように、GET メソッド（環境変数 “REQUEST_METHOD” が “GET”）の場合は CGI 変数は環境変数 “QUERY_STRING” の中に入っている。そしてその中身は、

name1=*var1*&*name2*=*var2*&...

というように “変数名=値” が “&” で繋がれた形式になっており、かつ “値” は URL エンコードされている。

ここまでわかっていれば自力で読み解くコードを書いてもよい。tr コマンドで “&” を改行に、“=” をスペースに代え、最初のスペースより右側〜行末までの文字列をレシピ 4.1（URL デコードする）に記したやり方でデコードするのだ。しかし、それを既に済ませたコマンドがあるので使わせてもらえばよいというわけだ。

「回答」で登場した 2 つのコマンドであるが、“cgi-name” はとりあえず CGI 変数文字列を扱いやすい形式に変換して一時ファイルに格納するためのもので、“nameread” は好きなタイミングでシェル変数等に取り出すためのものである。よって、前者は通常最初に一度だけ使うが、後者は必要な個所でその一時ファイルと共に毎回使うものである。

補足

ここで補足しておきたい事項が 3 つある。

■環境変数を echo ではなく printf で受け取る理由

環境変数 “'QUERY_STRING' を echo で受けず、なぜわざわざ printf で受けているのか。理由は、万が一環境変数に “-n” という文字列が来た場合でも誤動作しないようにするためである。

通常は起こりえないのだが、**外部からやってくる情報なので素直に信用してはいけない**というのが Web アプリケーション制作における鉄則だからである。

■値としての改行の扱い

受け取ったデータの中に改行文字（<CR><LF> 等）が含まれていた場合、cgi-name コマンドは “¥n” という 2 文字に変換する。詳細はマニュアルページ[*5]を参照されたい。

■GET メソッドか POST メソッドかを判定する

到来する CGI 変数データが POST メソッドでやってくるのか GET メソッドでやってくるのか決まっていない場合もあるだろう。そのような時は環境変数 REQUEST_METHOD の値が “GET” か “POST” かで分岐させればよい。その値が “POST” だった場合には次のレシピ 4.4（CGI 変数の取得 (POST メソッド編)）に示す方法で受け取ればよい。

参照

→レシピ 4.1（URL デコードする）
→レシピ 4.4（CGI 変数の取得 (POST メソッド編)）

レシピ 4.4　CGI 変数の取得（POST メソッド編）

問題

Web ブラウザーから送られてくる CGI 変数を読み取りたい。
ただし POST メソッド（環境変数 “REQUEST_METHOD” が “POST” の場合）である。

回答

基本的にはレシピ 4.3（CGI 変数の取得 (GET メソッド編)）と同じである。よってまず 2 つのコマンド（“cgi-name” 及び “namread”）をダウンロード[*6]する。

ここでも例をあげて説明しよう。今、Web ブラウザーが次のような HTML に基づいて CGI 変数を送ってくるものとする。

[*5] https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_cgi-name
[*6] https://github.com/321516/Open-usp-Tukubai/blob/master/COMMANDS.SH/cgi-name、及び https://github.com/321516/Open-usp-Tukubai/blob/master/COMMANDS.SH/nameread にアクセスし、そこにあるソースコードをコピー&ペーストしてもよいし、あるいは “RAW” と書かれているリンク先を「名前を付けて保存」してもよい。

■Web ブラウザーが送信する元になる HTML

```
<form action="form.cgi" method="POST">
  <dl>
    <dt>名前</dt>
      <dd><input type="text" name="fullname" value="" /></dd>
    <dt>メールアドレス</dt>
      <dd><input type="text" name="email" value="" /></dd>
  </dl>
  <input type="submit" name="post" value="送信" />
</form>
```

これを取得するためのシェルスクリプトは次のようになる。

■前述のフォームから送られてきた CGI 変数を受け取るシェルスクリプト（form.cgi）

```
#! /bin/sh

Tmp=/tmp/${0##*/}.$$  # 一時ファイルの元となる名称

dd bs=${CONTENT_LENGTH:-0} count=1 |
cgi-name                           > $Tmp-cgivars  # 正規化し、一時ファイルに格納

fullname=$(nameread fullname $tmp-cgivars)         # CGI 変数"fullname"を取り出す
email=$(nameread email $tmp-cgivars)               # CGI 変数"email"を取り出す

(ここで何らかの処理)

rm -f $Tmp-*                                       # 用が済んだら一時ファイルを削除
```

GET メソッドとの唯一の違いは、読み出す元が環境変数ではなく標準入力に代わったことだ。プログラム上ではそれに対応するため、dd コマンドを用いるようになった点のみが異なっている。

解説

基本的な解説はレシピ 4.3（CGI 変数の取得 (GET メソッド編)）で済ませてあるので、同じことに関しては省略する。ここでは POST メソッドで異なる点についてのみ述べる。

先程も述べたように、POST は CGI 変数の格納されている場所が環境変数ではなく標準入力であるという点が唯一異なる。標準入力から受け取るなら cat コマンドでもいいような気がするが、安全を期して dd コマンドを使うべきだ。

理由は、環境によっては運が悪いと標準入力からのデータを受け取るのに失敗して cat コマンド実行で止まってしまう恐れがあるからだ。CGI 変数文字列のサイズ（環境変数 CONTENT_LENGTH）が 0 である場合は読み取る必要がないのだが、環境によっては 0 なのに読もうとすると止まってしまうことがあるようだ。そのためにこのようなやり方を推奨している。

参照

→レシピ 4.3（CGI 変数の取得 (GET メソッド編)）
→レシピ 4.5（Web ブラウザーからのファイルアップロード）

レシピ 4.5　Web ブラウザーからのファイルアップロード

問題

Web ブラウザーからファイルをアップロードして、受け取れるようにしたい。

回答

CGI 変数の受け取りと同様に、アップロードされてきたファイルを受け取るのに便利なコマンド（“mime-read”）が Open usp Tukubai で提供されており、これらを POSIX の範囲で書き直したものが 321516 氏によって提供されているので、まずこれをダウンロード[*7]する。

GET、POST のレシピと同様に例をあげて説明しよう。今、Web ブラウザーが次のような HTML に基づいて CGI 変数を送ってくるものとする。

■Web ブラウザーが送信する元になる HTML

```
<form action="form.cgi" method="POST" enctype="multipart/form-data">
  <dl>
    <dt>証明写真ファイル</dt>
      <dd><input type="file" name="photo" /></dd>
    <dt>写っている人の名前</dt>
      <dd><input type="text" name="fullname" value="" /></dd>
  </dl>
  <input type="submit" name="post" value="送信" />
</form>
```

ファイルアップロード時は一般的に、POST メソッドで multipart/form-data 形式を用いるが、これを取得するためのシェルスクリプトは次のようになる。

[*7] https://github.com/321516/Open-usp-Tukubai/blob/master/COMMANDS.SH/mime-read にアクセスし、そこにあるソースコードをコピー&ペーストしてもよいし、あるいは “RAW” と書かれているリンク先を「名前を付けて保存」してもよい。

■前述のフォームから送られてきた CGI 変数を受け取るシェルスクリプト（form.cgi）

```
#! /bin/sh

Tmp=/tmp/${0##*/}.$$                                # 一時ファイルの元となる名称

dd bs=${CONTENT_LENGTH:-0} count=1 > $Tmp-cgivars  # そのまま一時ファイルに格納

mime-read photo $Tmp-cgivars > $Tmp-photofile      # CGI 変数"photo"をファイルとして保存

# アップロードされたファイル名を取り出すなら例えばこのようにする
filename=$(mime-read -v $Tmp-cgivars                                              |
           grep -i '^[0-9]¥+[[:blank:]]*Content-Disposition:[[:blank:]]*form-data;' |
           grep '[[:blank:]]name="photo"'                                         |
           head -n 1                                                              |
           sed 's/.*[[:blank:]]filename="¥([^"]*¥)".*/¥1/'                         |
           tr '/"' '--'                                                           )

fullname=$(mime-read fullname $Tmp-cgivars)        # CGI 変数"fullname"をファイルとして保存

(ここで何らかの処理)

rm -f $Tmp-*                                       # 用が済んだら一時ファイルを削除
```

通常の POST メソッドの場合と違い、到来した CGI 変数データは何も加工せずにそのまま一時ファイルに置き、ファイルや CGI 変数が欲しいたびに mime-read コマンドを使う。

また、ファイルに関してはアップロード時のファイル名も取得可能だ。mime-read コマンドの-v オプションを付けると、MIME ヘッダーを返すようなるため、UNIX コマンドを駆使して取り出せばよい。

解説

HTTP でのファイルアップロードは一般的に、POST メソッドを用いて行うため、レシピ 4.4（CGI 変数の取得 (POST メソッド編)）と同様に標準入力を読み出せばいいのだが、multipart/form-data という MIME ヘッダー付のフォーマットで到来する点が異なる。

先程の HTML であれば、次のようなデータが送られてくる。

■前述の HTML から送られてくるデータの例

```
--751A8F78020934B141231A1121CD31EF
Content-Disposition: form-data; name="photo"; filename="D:¥work¥komei.jpg"
Content-Type: image/jpeg

(ここに JPEG ファイルの中身……
      :
      :
      :
……)
--751A8F78020934B141231A1121CD31EF
Content-Disposition: form-data; name="fullname"

諸葛孔明
--751A8F78020934B141231A1121CD31EF
```

ハイフンで始まる行は各々の CGI 変数データセクションの境界を表しており、後ろのランダムな数字をもっ

て、データの中身と重複しないようにしている。1 つのセクションはヘッダー部とボディー部からなり、セクション内の最初の空行で仕切られる。従って変数名やファイル名はヘッダー部から取り出し、値はボディー部をそのまま取り出せばよい。

ボディー部分は、基本的に何のエンコードもされないためバイナリーデータである。これを取り出すのは一工夫必要だ、AWK は NULL（<0x00>）を含んでいるとそこを行末とみなして以降の行末までの文字列が取り出せない[*8]ので、取り出すのには使えないからだ。

ではどうするかというと、目的のデータのボディー部分は何行目から始まって何行目まで終わるのかを先に数える。そしてその区間を head コマンドと tail コマンドで抽出し、データ終端についている改行文字を消すのだ。この作業をコマンド化したものが、POSIX の範囲で書き直した mime-read コマンドなのである。

参照

→レシピ 4.3（CGI 変数の取得 (GET メソッド編)）
→レシピ 4.5（Web ブラウザーからのファイルアップロード）
→レシピ 1.6（改行無し終端テキストを扱う）

レシピ 4.6　Ajax で画面更新したい

問題

Web アプリ制作で、画面全体を更新せず、Ajax を用いて部分更新したい。
ただ、JavaScript ライブラリーは懲り懲りだ。prototype.js は下火になってしまったし、jQuery も頻繁にアップデートを繰り返していて、追いかけるのが大変だし。

回答

POSIX 原理主義を貫く意義を省みれば、クライアント上の JavaScript でも W3C で勧告されていない範囲のライブラリーを使うべきではない。従ってここでも自力で行う方法を紹介する。

POSIX 原理主義者的 Ajax チュートリアル

では、簡単な Ajax 利用 Web アプリを作ってみよう。HTML フォームのボタンを押すたび Ajax でサーバーに現在時刻を問い合わせ、時刻欄を書き換えるというものだ。リストは 3 つ必要だ。まずは HTML から。

[*8] GNU 版 AWK は取り出せるのだが。

■CLOCK.HTML

```
<html>
  <head>
    <title>Ajax Clock</title>
    <style type="text/css">
      #clock {border: 1px solid;width 20em}
    <script type="text/javascript" src="CLOCK.JS"></script>
  </head>
  <body onload="update_clock()">
    <h1>Ajax Clock</h1>
    <div id="clock">
      <dl>
        <dt>Date:</dt><dd>0000/00/00</dd>
        <dt>Time:</dt><dd>00:00:00</dd>
      </dl>
    </div>
    <input type="button" value="update" onclick="update_clock()">
  </body>
</html>
```

次に Ajax 通信時にサーバー上でレスポンスを返す CGI スクリプト。Web ブラウザーから Ajax として呼ばれた際、現在時刻を取得して前述 HTML の 〜 の中身を生成して返すというものだ。

この CGI スクリプトが **XML や JSON ではなく、部分的な HTML を返している**という点も JavaScript ライブラリー依存から脱するための重要な工夫だ。

■CLOCK.CGI

```
#! /bin/sh

datetime=$(date '+%Y/%m/%d %H:%M:%S')
cat <<HTTP_RESPONSE
Content-Type: text/html

      <dl>
        <dt>Date:</dt><dd>${datetime% *}</dd>
        <dt>Time:</dt><dd>${datetime#* }</dd>
      </dl>
HTTP_RESPONSE
exit 0
```

そして最後に、Ajax 通信を行う JavaScript(CLOCK.JS) の中身は次のとおりだ。

■CLOCK.JS

```
// 1.Ajax オブジェクト生成関数
// (IE、非 IE 共に XMLHttpRequest オブジェクトを生成するためのラッパー関数)
function createXMLHttpRequest(){
  if(window.XMLHttpRequest){return new XMLHttpRequest()}
  if(window.ActiveXObject){
    try{return new ActiveXObject("Msxml2.XMLHTTP.6.0")}catch(e){}
    try{return new ActiveXObject("Msxml2.XMLHTTP.3.0")}catch(e){}
    try{return new ActiveXObject("Microsoft.XMLHTTP")}catch(e){}
  }
  return false;
}

// 2.Ajax 通信関数
// (この Ajax 通信をしたい時にはこの関数を呼び出す)
function update_clock() {
  var url,xhr,to;
  url = '/PATH/TO/THE/CLOCK.CGI';
  xhr = createXMLHttpRequest();
  if (! xhr) {return;}
  to =  window.setTimeout(function(){xhr.abort()}, 30000); // 30 秒でタイムアウト
  xhr.onreadystatechange = function(){update_clock_callback(xhr,to)};
  xhr.open('GET' , url+'?dummy='+(new Date)/1, true);       // キャッシュ対策
  xhr.send(null); // POST メソッドの場合は、send() の引数として CGI 変数文字列を指定
}

// 3. コールバック関数
// (Ajax 通信が正常終了した時に実行したい処理を、この if 文の中に記述する)
function update_clock_callback(xhr,to) {
  var str, elm;
  if (xhr.readyState === 0) {alert(' タイムアウトです。');}
  if (xhr.readyState !== 4) {return;                      } // Ajax 未完了につき無視
  window.clearTimeout(to);
  if (xhr.status === 200) {
    str = xhr.responseText;
    elm = document.getElementById('clock');
    elm.innerHTML = str;
  } else {
    alert(' サーバーが不正な応答を返しました。');
  }
}
```

このように、コメントを除けば 40 行足らずの JavaScript コードで、Ajax が実装できてしまう。

この 3 つのコードを適宜 Web サーバーにアップロード (CLOCK.JS 内で指定している URL は適切に記述すること) し、Web ブラウザーで CLOCK.HTML を開いてみるとよい。update ボタンを押すたびに現在時刻に更新されるはずだ。

解説

世の中 JavaScript が流行っており、同時に、それを便利に使うためのライブラリーも実に様々なものが登場している。昔は prototype.js が流行ったが廃れ、トレンドは jQuery へ移りっている。しかし度重なるバージョンアップに追加モジュールがある。もうこうなると「一体どれを使えばよいのか???」、Ajax や JavaScript 初心者はまずそこから悩むことになる。そんなことに時間を費やすくらいなら前述のような 40 行足らずのコードを

理解し、コピー＆ペーストして使う方がよっぽど簡単ではなかろうか。

前述の JavaScript コードは XMLHttpRequest という Ajax のためのオブジェクトを使うためのコードだが、いくつかのポイントを押さえれば理解は簡単だ。

ポイント 1. XMLHttpRequest オブジェクト生成方法

IE（Internet Explorer）は他のブラウザーと違って少々クセがある。まずは XMLHttpRequest オブジェクトの生成方法が違う（オブジェクトの使い方は同じ）。IE でも最近のものは他と同様の方法で生成できるが、古い IE は ActiveX オブジェクトとして生成しなければならない。そこで、オブジェクトの生成を色々な手段で成功するまで試みるのが最初の関数 createXMLHttpRequest() である。これを使えばどのブラウザーで動かされるのかを気にせずオブジェクトが生成できる。

ポイント 2. キャッシュ回避テクニック

これまた IE 対策なのだが、IE は同じ内容で Ajax 通信を行うと 2 回目以降はキャッシュを見にいってしまい、実際の Web サーバーへはアクセスをしないという困ったクセがある。これを回避するテクニックが「キャッシュ対策」とコメントしてある行の記述だ。URL の後ろに、ダミーの CGI 変数を置き、その値として UNIX 時間 (のミリ秒単位を求める Date オブジェクトの利用) とすることで、アクセスする度、リクエスト内容が変わるようにしている。これで IE もキャッシュを使わなくなる。一応 XMLHttpRequest にはキャッシュを使わせないためのメソッドが用意されているのだが、これまたバージョンによって使い方が微妙に異なるので、CGI 変数を毎回替えるというこの原始的な方法が最も確実である。尚、POST メソッドの場合の CGI 変数は、その一つ後の send メソッドの引数として指定することになっているので注意。

XMLHttpRequest オブジェクトやその各種メソッドやプロパティーの使い方については、その名前で Web 検索してもらいたい。様々なページで解説されている。

ポイント 3. 無理に XML や JSON を使おうとしない

このサンプルコードのもう一つの特徴は、**Ajax でありながら XML をやりとりしていない**ことだ。だからといって、最近 XML の代わりに使われるようになってきた JSON も利用していない。サーバー側の CGI は時刻データを XML や JSON で表現したものを返しているのではなく、ハメ込まれる`<div>`タグの中身の部分 HTML ごと作ってしまっている。こうすると JavaScript 側は受け取った文字列を innerHTML プロパティーに代入するだけで済んでしまう。このようにしてサーバー側に部分 HTML の生成を任せてしまえば、ライブラリー無しの JavaScript 側で、苦労して HTML の DOM ツリーを操作するなどといった面倒な作業が要らなくなる。そのぶんサーバー側が大変になると思うかもしれないが、レシピ 4.8（HTML テーブルを簡単綺麗に生成する）で紹介している便利な mojihame コマンドを使えば、それほど大変なことにはならないのだ。

ただし困ったことに **IE8 は、`<select>`タグには innerHTML プロパティー代入ができない**というバグがある。これがやりたい場合は残念ながら XML や JSON を使うしかない。

参照

→ MDN（Mozilla Developer Network）サイトの XMLHttpRequest メソッド説明ページ[*9]
→レシピ 4.8（HTML テーブルを簡単綺麗に生成する）

[*9] `https://developer.mozilla.org/ja/docs/Web/API/XMLHttpRequest`

レシピ 4.7　シェルスクリプトでメール送信

問題

Web サーバーのアクセスログの集計結果を自動的に管理者と、Cc:付けて営業部長にメールで送りたい。ただ営業部長はエンジニアではないので、できれば日本語の件名と文面で送りたい。どうすればよいか。

回答

POSIX 標準の mailx コマンドを使って送れと言いたいところだが、Cc:で送ること、さらに日本語メールを送ることができない[*10]。

POSIX 原理主義を貫きたいところであるが、ここは涙を飲んでいくつかの非 POSIX 標準コマンドに頼る。詳細については次のチュートリアルを参照せよ。

Step(0/3) ー 必要な非 POSIX コマンド

表 4.1 に必要なコマンドの一覧を記す。非 POSIX 標準ではあるが、多くの環境に初めから入っている、もしくは容易にインストールできるものではあると思う。

表 4.1 日本語メールを送るために用いる非 POSIX コマンド

コマンド名	目的	備考
sendmail	Cc:や任意のヘッダー付、また日本語のメールを送るため	多くの UNIX 系 OS に標準で入っていたり、Postfix や qmail を入れても互換コマンドが/usr/sbin に入る。
nkf	日本語文字エンコード（特に Subject:欄）のため	パッケージとして提供されている OS も多く、ソースからインストールすることも可能（ver.2.1.2 以降推奨）

もし、Sujbect:ヘッダーや From:、Cc:、Bcc:ヘッダー等には日本語文字を使わないというのであれば nkf コマンドについては、POSIX 標準の iconv コマンドで代用することができる。

Step(1/3) ー 英数文字メールを送ってみる

まずは sendmail コマンドだけで済む内容のメールを送り、sendmail コマンドの使い方を覚えることにしよう。といっても、 `-i` と `-t` オプションさえ覚えれば OK。これらさえ知っていれば、他のものは覚えなくても大丈夫だ[*11]。

[*10] 任意のメールヘッダーを付けられず、マルチバイト文字を使っていることを示すヘッダーが付けられないため、受信先の環境によっては文字化けしてしまうから。

[*11] どうしても意味を知っておきたいという人は、sendmail コマンドの man を参照のこと。URL は例えば `http://www.jp.freebsd.org/cgi/mroff.cgi?subdir=man&lc=1&cmd=&man=sendmail&dir=jpman-10.1.2%2Fman&sect=0` である。

ここで肝心なことは、**一定の書式のテキスト作って、標準入出力経由で “sendmail -i -t” に流し込む**ということだけである。

ではまず、適当なテキストエディターで下記のテキストを作ってもらいたい。

■メールサンプル（mail1.txt）

```
From: <SENDER@example.com>
To: <RECIEVER@example.com>
Subject: Hello, e-mail!

Hi,can you see me?
```

メールテキストを作る時のお約束は、**ヘッダー部分のセクションと本文セクションの間に空行 1 つを挟む**ことだ。これを怠るとメールは送れない。

RECIEVER@example.com にはあなた本物のメールアドレスを書くように。それから *SENDER@example.com* にも、なるべく何か実際のメールアドレスを書いておいてもらいたい。あまりいい加減なものを入れると、届いた先で spam 判定されるかもしれないからだ。

「**Cc:や Bcc:も追記したら、Cc:や Bcc:でも送れるのか**」と想像するかもしれないが、そのとおりである。実験してみるといい。それができたら送信する。次のコマンド打つだけだ。

```
$ cat mail1.txt | sendmail -i -t ↵
$
```

コマンドを連打すれば、連打した数だけ届くはずだ。確かめてみよ。

step(2/3) ー 本文が日本語のメールを送ってみる

このドキュメントを読んでいるのは日常的に日本語を使う人々のはずだから、次は本文が日本語のメールを送ってみることにする。

まずは、日本語の本文を交えたメールテキストを作る。

■メールサンプル（mail2.txt）

```
From: <SENDER@example.com>
To: <RECIEVER@example.com>
Subject: Hello, e-mail!
Content-type: text/plain; charset=ISO-2022-JP

やぁ、これ読める?
```

本文に日本語文字が入った他に、ヘッダー部に `Content-type: text/plain; charset=ISO-2022-JP` を追加した。実際のところ、今となってはこれが無くても読める環境が多いのだが、本文が日本語文字コードを使っていることを示すための約束事であるので付けるべきである。

「このテキストは UTF-8 で書いてるんだけど」と心配するかもしれないが大丈夫、これから nkf コマンドを使って JIS コードに変換するのだ。

今度は、途中に “`nkf -j`”（JIS コードに変換）を挿んでメール送信する。

```
$ cat mail1.txt | nkf -j | sendmail -i -t ↵
$
```

きちんと文字化けせずにメールが届いたことを確認してもらいたい。

step(3/3) ー 件名や宛先も日本語化したメールを送る

From:や To:はまぁ許せるとしても、Subject:(件名) には日本語を使いたい。そこで最後は、件名や宛先も日本語化する送信方法を説明する。

まず、メールヘッダーには生の JIS コード文字列が置けず、置いた場合の動作は保証されないということを知らなければならない。ヘッダーはメールに関する制御情報を置く場所なので、あまり変な文字を置いてはいけないのだ。だが Base64 エンコードもしくは quoted-string エンコードしたものであれば置いてもよいことになっている。そこで、Base64 エンコードを使うやり方を説明する。現在殆どのメールでは Base64 エンコードが用いられている。

メールで使える Base64 エンコード済 JIS コード文字列の作り方

例として、送りたいメールの件名は「ハロー、e-mail!」、そして宛先は「あなた」ということにしてみる。

冒頭で好きな文字列を書いた echo コマンドを入れて、xargs と nkf コマンドに流すだけだ。

■「ハロー、e-mail!」をエンコード

```
$ echo -n ’ハロー、e-mail!’                | ↵←改行コードが入らぬように-n オプションを付ける
> nkf -jMB                                 | ↵←オプションは “-jMB”
> xargs printf ’=?ISO-2022-JP?B?%s?=¥n’ ↵←文字列両端をエンコード済を意味する表記で挟む
=?ISO-2022-JP?B?GyRCJU8lbSE8ISIbKEJlLW1haWwh?=
$
```

■「あなた」をエンコード

```
$ echo -n ’あなた’                         | ↵
> nkf -jMB                                 | ↵
> xargs printf ’=?ISO-2022-JP?B?%s?=¥n’ ↵
=?ISO-2022-JP?B?GyRCJCIkSiQ/GyhC?=
$
```

コード中のコメントにも書いたが、ポイントは次の 3 つである。

1. 最初に流す文字列には余計な改行をつけぬこと（echo の-n オプションを使うなどして）
2. nkf コマンドを使うが、オプションは-jMB とする（JIS 化してさらに Base64 エンコード）
3. 生成された文字列の左端に “=?ISO-2022-JP?B?” を、右端に “?=” を付加

送信してみる

送るには、上記の作業で生成された文字列をメールヘッダーにコピペすればよい。早速メールテキストを作ってみよう。

■メールサンプル（mail3.txt）

```
From: <SENDER@example.com>
To: =?ISO-2022-JP?B?GyRCJCIkSiQ/GyhC?= <RECIEVER@example.com>
Subject: =?ISO-2022-JP?B?GyRCJU8lbSE8ISIbKEJlLW1haWwh?=
Content-type: text/plain; charset=ISO-2022-JP

やぁ、これ読める?
```

しかし、これを先程の nkf+sendmail コマンドに送ると失敗してしまう。理由は、nkf コマンドがせっかく作った Base64 エンコードを解いてくれてしまうからなのだ。これを回避するため、テンポラリーファイルを介してシェルスクリプトで送る。

■第一引数で指定されたファイルをメール送信するシェルスクリプト

```
#! /bin/sh

# ヘッダーセクションはそのまま書き出す
awk '{print} length($0)==0{exit}' "$1" >  /tmp/${0##*/}.$$.txt

# 本文部分は nkf で JIS エンコードして、追記する
startofbody=$(awk 'END{print NR+1}' /tmp/${0##*/}.$$.txt)
cat "$1"                                |
tail -n +$startofbody                   |
nkf -j                                  >> /tmp/${0##*/}.$$.txt

# メール送信
cat /tmp/${0##*/}.$$.txt |
nkf -j                   |
sendmail -i -t

# 一時ファイル削除
rm /tmp/${0##*/}.$$.txt
```

宛先、件名共に日本語文字列になったメールが届いたか確かめてもらいたい。

これで問題文にあった「Cc:付けて営業部長に日本語メール」もできるようになるだろう。

レシピ 4.8　HTML テーブルを簡単綺麗に生成する

問題

AWK 等で生成したテキスト表の内容を HTML で表示したい。それこそ AWK の printf() 関数等を使って HTML コードを生成するループを書けばいいのはわかるが、それだとプログラム本体と HTML デザインがごっちゃになってしまって、メンテナンス性が悪い。何かいい方法はないか？

回答

シェルスクリプト開発者向けコマンドセット Open usp Tukubai に収録されている “mojihame” というコマンドを使うとその悩みは綺麗に解消できる。このコマンドを使えば、HTML の一部分（例えば <tr>〜</tr>）をレコードの数だけ繰り返すという指示を、プログラム本体ではなく HTML テンプレートの中で指定することができるようになるからだ。

利用を検討している人は、mojihame コマンドをダウンロード[*12]し、次のチュートリアルを参考にしてもらいたい。

mojihame コマンドチュートリアル

筆者が制作した、東京メトロのオープンデータに基づく列車在線情報表示アプリケーション「メトロパイパー」[*13]を例にしたチュートリアルを記す。

このアプリケーションは、「知りたい駅」と「行きたい方面駅」の 2 つの駅を指定すると、前者の駅周辺のリアルタイム列車の在線状況を表示するというものである。

その 1. 単純繰り返し

メトロパーパーでは mojihame コマンドを二つのシェルスクリプトで活用しており、そのうちの 1 つは「知りたい駅」や「行きたい方面駅」の駅名選択肢の表示だ。これらの選択肢は <select> タグを使って描画しているが、その中の <option> タグが各駅に対応していて、<option>〜</option> の区間を駅の数だけ繰り返して表示したいわけである。

最初に HTML テンプレートファイル[*14]を用意する。

■HTML テンプレート MAIN.HTML（駅選択肢部分を抜粋）

```
      :
  <select id="from_snum" name="from_snum" onchange="set_snum_to_tosnum()" >
    <!-- FROM_SELECT_BOX -->
    <option value="-">ー</option>
    <!-- FROM_SNUM_LIST
    <option value="%1">%1 : %2 線-%3 駅</option>
         FROM_SNUM_LIST -->
    <!-- FROM_SELECT_BOX -->
  </select>
     :
```

色々 HTML コメントが付いているが、“FROM_SELECT_BOX” という区間と、“FROM_SELECT_BOX” という区間があるのに注目してもらいたい。“FROM_SELECT_BOX” は、テンプレートファイル “MAIN.HTML” から、mojihame による処理を行うために <select> タグの内側を抽出するために付けた文字列である。具体的には sed コマンドで行う（後述）。一方 “FROM_SELECT_BOX” は、mojihame コマンドに対して繰り返し区間の始まりと終わりを示すためのものである。先程の “FROM_SELECT_BOX” で取り出した区間には、デフォルト選択肢 “<option value="-">ー</option>” が含まれているがこれは繰り返し区間には含めさせないために用意して

[*12] https://github.com/321516/Open-usp-Tukubai/blob/master/COMMANDS.SH/han にアクセスし、そこにあるソースコードをコピー&ペーストしてもよいし、あるいは “RAW” と書かれているリンク先を「名前を付けて保存」してもよい。

[*13] http://metropiper.com/

[*14] HTML 全体を見たいという人は、https://github.com/ShellShoccar-jpn/metropiper/blob/master/HTML/MAIN.HTML を参照されたい。

ある。

そして注目すべきは、中にある “%1”、“%2” といったマクロ文字列だ。これらが実際の駅ナンバーや路線名、駅名に置換されていく。

ではその置換対象となる駅データを見てみよう。

■与える選択肢テキスト（抜粋）

```
C01 千代田 代々木上原
C02 千代田 代々木公園
C03 千代田 明治神宮前〈原宿〉
        :
        :
Z14 半蔵門 押上〈スカイツリー前〉
```

左の列から、駅ナンバー、路線名、駅名という構成になっているが、これが先程の “%1”、“%2”、“%3” をそれぞれ置き換えていくことになる。

そして実際に置き換えを実施しているプログラム[*15]がこれだ。

■選択肢生成プログラム GET_SNUM_HTMLPART.AJAX.CGI（抜粋）

```
# --- 部分 HTML のテンプレート抽出 -------------------------------------
cat "$Homedir/TEMPLATE.HTML/MAIN.HTML"        |
sed -n '/FROM_SELECT_BOX/,/FROM_SELECT_BOX/p' |
sed 's/一/選んでください/'                     > $Tmp-htmltmpl

# --- HTML 本体を出力 -------------------------------------------------
cat "$Homedir/DATA/SNUM2RWSN_MST.TXT"     |
# 1:駅ナンバー (sorted) 2:路線コード 3:路線名 4:路線駅コード
# 5:駅名 6:方面コード (方面駅でない場合は"-")
grep -i "^$rwletter"                      |
awk '{print substr($1,1,1),$0}'           |
sort -k1f,1 -k2,2                         |
awk '{print $2,$4,$6}'                    |
uniq                                      |
# 1:駅ナンバー (sorted) 2:路線名 3:駅名    #
mojihame -lFROM_SNUM_LIST $Tmp-htmltmpl -
```

先程述べたように、まずコメント “FROM_SELECT_BOX” の区間をテンプレートファイルから sed コマンドで抽出し、一時ファイル “$Tmp-htmltmpl” に格納している。そして 2 番目の cat コマンドから uniq コマンドまでの間に、先程の 3 列構成のデータを生成し、mojihame コマンドに渡しているのである。mojihame コマンドでは、-l オプションが単純繰り返しの際に用いるものであり、そのオプションの直後（スペース無し）に繰り返し区間を示す文字列を指定する。mojihame コマンドの詳しい使い方は man ページ[*16]を参照してもらいたい。

結果としてこの mojihame コマンドは次のようなコードを出力する。

[*15] HTML 全体を見たいという人は、
https://github.com/ShellShoccar-jpn/metropiper/blob/master/CGI/GET_SNUM_HTMLPART.AJAX.CGI を参照されたい。

[*16] https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_mojihame

```
    <!-- FROM_SELECT_BOX -->
    <option value="-">ー</option>
    <option value="C01">C01 : 千代田線-代々木上原駅</option>
    <option value="C02">C02 : 千代田線-代々木公園駅</option>
    <option value="C03">C03 : 千代田線-明治神宮前〈原宿〉</option>
        :
    <option value="Z14">Z14 : 半蔵門線-押上〈スカイツリー前〉</option>
    <!-- FROM_SELECT_BOX -->
```

このプログラムのファイル名に “AJAX” と書かれていることから想像できるように、このプログラムは Ajax として動くようになっている。具体的には、クライアント側では上記の部分 HTML を受け取った後、innerHTML プロパティーを使って <select> タグエレメントに流し込んでいる。この手法は、レシピ 4.6（Ajax で画面更新したい）で記した方法そのものである。

その 2. 階層を含む繰り返し

メトロパイパーにてもう一つ mojihame コマンドを利用しているのは、実際の在線情報欄の HTML を作成する “GET_LOCINFO.AJAX.CGI” というシェルスクリプトである。こちらではもう少し高度な使い方をしている。

まず、mojihame コマンドを使って出来上がった完成画面（図 6.1）を見てもらいたい。

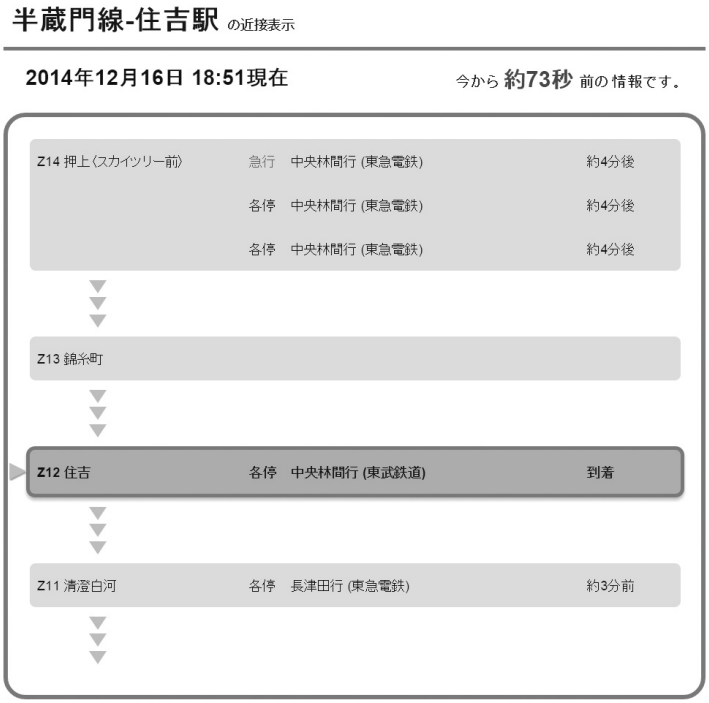

図 4.1 メトロパイパーの在線情報

この時押上駅には列車が 3 本在線しており、押上駅の欄が他の駅や駅間よりも広がっている。この画面を作成するにあたっては、押上駅、駅間（押上—錦糸町）、錦糸町駅、駅間（錦糸町—住吉）、……という繰り返しをしているが、さらに各駅の中では列車が複数いればそこでも複数回の繰り返しをするというようにして階層化され

た繰り返しを行っているのだ。

具体的には、プログラム内部で、一旦次のようなデータが生成される。

■生成される在線情報データ（抜粋）

```
Z140 odpt.Station:TokyoMetro.Hanzomon.Oshiage 押上〈スカイツリー前〉 odpt.TrainType:TokyoMet…
Z140 odpt.Station:TokyoMetro.Hanzomon.Oshiage 押上〈スカイツリー前〉 odpt.TrainType:TokyoMet…
Z140 odpt.Station:TokyoMetro.Hanzomon.Oshiage 押上〈スカイツリー前〉 odpt.TrainType:TokyoMet…
Z135 - - - - - - - - - 5
Z130 odpt.Station:TokyoMetro.Hanzomon.Kinshicho 錦糸町 - - - - - - - 10
Z125 - - - - - - - - - 15
Z120 odpt.Station:TokyoMetro.Hanzomon.Sumiyoshi 住吉 odpt.TrainType:TokyoMetro.Local 各停 od…
Z115 - - - - - - - - - 25
  :
```

このデータは駅名（駅コード）とそこに在線する列車（列車コード）の並んだ表であるが、同じ駅に複数の列車が在線している場合には、同じ駅のレコードが繰り返される仕様になっている。従って、このデータからは押上に 3 つの列車がいることがわかる。

そしてこのデータを受ける HTML テンプレート[*17]は次のとおりである。

■HTML テンプレート LOCTABLE_PART.HTML（抜粋）

```
  :
<!-- /LC_HEADER -->
<!-- LC_ITERATION-1（繰り返し区間メイン…駅）-->
    <div class="%1 %2 clearfix">
  <!-- LC_ITERATION-2（繰り返し区間サブ…車両）-->
      <div class="station_name"><a href="%10" target="_blank">%3 %4</a></div>
      <div class="train_info">
        <div class="train_assort %5">%6</div>
        <div class="train_for">%7 %8</div>
      </div>
      <div class="approach_time">%9</div>
  <!-- /LC_ITERATION-2 -->
    </div>
<!-- /LC_ITERATION-1 -->
<!-- LC_FOOTER -->
  :
```

これを先程のシェルスクリプト “GET_LOCINFO.AJAX.CGI”[*18]の次の行で、同様にしてハメんでいる。

```
mojihame -hLC_ITERATION $Homedir/TEMPLATE.HTML/LOCTABLE_PART.HTML > $Tmp-loctblhtml0
```

階層化された繰り返し対応させるため、今度は “-h” というオプションを用いている。これも詳細は man ページを参照してもらいたい。

結果、生成された HTML テキストが次のものである。

[*17] HTML 全体を見たいという人は、
https://github.com/ShellShoccar-jpn/metropiper/blob/master/TEMPLATE.HTML/LOCTABLE_PART.HTML を参照されたい。

[*18] https://github.com/ShellShoccar-jpn/metropiper/blob/master/TEMPLATE.HTML/LOCTABLE_PART.HTML 参照

■列車在線情報をハメ込んで生成された HTML コード（抜粋、右端にループの範囲を記している）

```
<!--/LC_HEADER -->
    <div class="station near clearfix">                                          ↑
      <div class="station_name">Z14 押上〈スカイツリー前〉</div>                  ↑   |
      <div class="train_info">                                                   |   |
        <div class="train_assort odpt.TrainType:TokyoMetro.Express">急行</div> 列   |
        <div class="train_for">中央林間行 (東急電鉄)</div>                       車   |
      </div>                                                                     |   |
      <div class="approach_time">約 4 分後</div>                                 ↓   |
      <div class="station_name">  </div>                                         ↑   |
      <div class="train_info">                                                   |   |
        <div class="train_assort odpt.TrainType:TokyoMetro.Local">各停</div>   列  駅
        <div class="train_for">中央林間行 (東急電鉄)</div>                       車   |
      </div>                                                                     |   |
      <div class="approach_time">約 4 分後</div>                                 ↓   |
      <div class="station_name">  </div>                                         ↑   |
      <div class="train_info">                                                   |   |
        <div class="train_assort odpt.TrainType:TokyoMetro.Local">各停</div>   列   |
        <div class="train_for">中央林間行 (東急電鉄)</div>                       車   |
      </div>                                                                     |   |
      <div class="approach_time">約 4 分後</div>                                 ↓   |
    </div>                                                                           ↓
    <div class="between near clearfix">                                          ↑
      <div class="station_name"><a href="#" target="_blank">  </a></div>         ↑   |
      <div class="train_info">                                                   |   |
        <div class="train_assort  "></div>                                      列  駅
        <div class="train_for">  </div>                                         車   |
      </div>                                                                     |   |
      <div class="approach_time"></div>                                          ↓   |
    </div>                                                                           ↓
    <div class="station near clearfix">                                          ↑
      <div class="station_name">Z13 錦糸町</div>                                 ↑   |
      <div class="train_info">                                                   |   |
        <div class="train_assort  "></div>                                      列  駅
        <div class="train_for">  </div>                                         車   |
      </div>                                                                     |   |
      <div class="approach_time"></div>                                          ↓   |
    </div>                                                                           ↓
        :
<!--LC_FOOTER -->
```

このようにして、mojihame コマンドを使えば HTML テンプレートとプログラムを別々に管理することができるので、デザイン変更時のメンテナンスも容易であるし、何より Web デザイナーとの協業がとても楽なのだ。

参照

→ mojihame man ページ[*19]
→レシピ 4.6（Ajax で画面更新したい）
→「メトロパイパー」ソースコード[*20]

レシピ 4.9　シェルスクリプトおばさんの手づくり Cookie（読み取り編）

問題

クライアント（Web ブラウザー）が送ってきた Cookie 情報を、シェルスクリプトで書いた CGI スクリプトで読み取りたい。

回答

Cookie 文字列は CGI 変数とよく似たフォーマットで、しかも環境変数で渡ってくるのでレシピ 4.3（CGI 変数の取得 (GET メソッド編)）がほぼ流用できる。しかしながら若干の相違点があるので、具体例を示しながら説明する。

例として掲示板の Web アプリケーションの場合を考えてみる。投稿者名とメールアドレスをそれぞれ “name”、“email” という名前で Cookie に保存していたとすると、それを取り出すには次のように書けばよい。

■掲示板の投稿者名と e-mail を Cookie から取り出す

```
#! /bin/sh

Tmp=/tmp/${0##*/}.$$                                # 一時ファイルの元となる名称

printf '%s' "${HTTP_COOKIE:-}" |
sed 's/&/%26/g'                |
sed 's/[;, ]¥{1,¥}/¥&/g'       |
sed 's/^&//; s/&$//'           |
cgi-name                       > $Tmp-cookievars    # 正規化し、一時ファイルに格納

name=$(nameread name $tmp-cookievars)               # Cookie 変数"name"を取り出す
email=$(nameread email $tmp-cookievars)             # Cookie 変数"email"を取り出す

(ここで何らかの処理)

rm -f $Tmp-*                                        # 用が済んだら一時ファイルを削除
```

GET メソッドとの違いは、

- 読み取る環境変数は “QUERY_STRING” ではなく “HTTP_COOKIE”
- 変数の区切り文字は “&” ではなく “;’

[*19] https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_mojihame
[*20] https://github.com/ShellShoccar-jpn/metropiper

の 2 つである。前述のシェルスクリプトの、printf 行は環境変数が替わっており、その下にある 3 つの sed 行は Cookie 変数文字列のフォーマットを CGI 変数文字列のフォーマットに変換し、cgi-name コマンドに流用するために追加したものである。

解説

Cookie のフォーマットについては RFC 6265 で詳しく定義さてれいるが、次のような文字列でやってくる。

name1=*var1*； *name2*=*var2*； ...

このルールさえわかれば自力でやるもの難しくはないが、CGI 変数文字列とよく似ているので “cgi-name” コマンドと “nameread” コマンドで処理できるように変換するのが簡単だ。

参照

→ RFC 6265 文書[*21]
→レシピ 4.3（CGI 変数の取得 (GET メソッド編)）
→レシピ 4.10（シェルスクリプトおばさんの手づくり Cookie(書き込み編)）

レシピ 4.10　シェルスクリプトおばさんの手づくり Cookie（書き込み編）

問題

掲示板 Web アプリケーションを作ろうと思う。投稿者の名前と e-mail アドレスを Cookie で、クライアント（Web ブラウザー）に 1 週間覚えさせたい。

回答

順を追っていけばシェルスクリプトでも手づくり（POSIX の範囲）で Cookie が焼ける（Cookie ヘッダーを作れる）し、クライアントから読み取ることもできる。しかしやることがたくさんあるので、POSIX の範囲で実装した “mkcookie” コマンド[*22]をダウンロードして使うことにする。

そして例えば、名前（name）とメールアドレス（email）を Cookie に覚えさせる CGI スクリプトであれば、次のように書く。

[*21] http://tools.ietf.org/html/rfc6265
[*22] https://github.com/ShellShoccar-jpn/misc-tools/blob/master/mkcookie

■掲示板で名前とメールアドレスを Cookie に覚えさせる CGI スクリプト（bbs.cgi）

```
#! /bin/sh

Tmp=/tmp/${0##*/}.$$                    # 一時ファイルの元となる名称

# （名前とメールアドレスを設定するための何らかの処理）

# "変数名 + スペース 1 文字 + 値" で表現された元データファイルを作成
cat <<-FOR_COOKIE > $Tmp-forcookie
    name $name
    email $email
FOR_COOKIE

# Cookie 文字列を作成
cookie_str=$(mkcookie -e +604800 -p /bbs -s Y -h Y $Tmp-forcookie)
             # -e +604800: 有効期限を 604800 秒後（1 週間後）に設定
             # -p /bbs   : サイトの/bbs ディレクトリー以下で有効な Cookie とする
             # -s Y      : Secure フラグを付けて、SSL 接続時以外には読み取れないようにする
             # -h Y      : httpOnly フラグを付けて、JavaScript には拾わせないようにする

# HTTP ヘッダーを出力
cat <<-HTTP_HEADER
    Content-Type: text/html$cookie_str

HTTP_HEADER

# （ここで HTML のボディー部分を出力）

rm -f $Tmp-*                            # 用が済んだら一時ファイルを削除
```

mkcookie コマンドに渡す変数は、1 変数につき 1 行で

変数名 < 半角スペース 1 文字 > 値

という書式にして作る。変数名と値の間に置く半角スペースは 1 文字にすること。もし 2 文字にすると 2 文字目は値としての半角スペースとみなすので注意。

mkcookie コマンドのオプションについては、“`--help`” オプションなどで表示される Usage を参照されたい。RFC 6265 で定義されている属性に対応しているのですぐわかるだろう。

最後に、ここで出来上がった Cookie 文字列は、出力しようとしている他の HTTP ヘッダーに付加して送る。注意すべき点が 1 つある。**mkcookie コマンドは、先頭に改行を付ける仕様になっている**ので、前述の例のように他のヘッダー（例えば Content-Type）の行末に付加し、単独の行とはしないよう気を付けなければならないということだ。

解説

クライアントに Cookie を送るためにはまず、Cookie 文字列がどんな仕様になっているかを知る必要がある。そこで具体例を示そう。まず、次のような条件があるとする。

- 投稿者の名前（name）は、「6 号さん」
- 投稿者のメールアドレス（email）は、“`6go3@example.com`”
- 有効期限は、現在（2014/12/30 日 10:20:30 とする）から 1 週間後

- サイトの “/bbs” ディレクトリー以下で有効
- “example.com” というドメインでのみ有効
- Secure フラグ（SSL でアクセスしている時のみ）有効
- httpOnly フラグ（JavaScript には取得させない）有効

この時に生成すべき Cookie 文字列は次のとおりだ。

```
Set-Cookie: name=6%E5%8F%B7%E3%81%95%E3%82%93; expires=Tue, 06-Jan-2015 19:20:30 GMT; path=/bbs;
 domain=example.com; Secure; HttpOnly
Set-Cookie: email=6go3%40example.com; expires=Tue, 06-Jan-2015 19:20:30 GMT; path=/bbs; domain=e
xample.com; Secure; HttpOnly
```

つまり、“Set-Cookie:” という名前の HTTP ヘッダーを用意し、そこに、変数名=値

- 変数名=値（必須）
- `expires=`有効期限の日時（RFC 2616 Sec3.3.1 形式、省略可）
- `path=`*URL* 上で使用を許可するディレクトリー（省略可）
- `domain=`*URL* 上で使用を許可するドメイン（省略可）
- `Secure`（省略可）
- `HttpOnly`（省略可）

という各種プロパティーを、カンマ区切りで付けていく。もし送りたい Cookie 変数が複数ある場合は、1 つ 1 つに “Set-Cookie:” 行をつけ、expires 以降のプロパティーは同じものを使えばよい。

このような Cookie 文字列を生成するにあたっては、ここまで紹介してきたレシピのうちの 2 つを活用する。値を URL エンコードするにはレシピ 4.2（URL エンコードする）、有効日時の計算にはレシピ 3.3（シェルスクリプトで時間計算を一人前にこなす）だ。有効日時は、“RFC 2616 Sec3.3.1 形式” ということになっているが、その形式を作るには次のコードで可能だ。

```
TZ=UTC+0 date +%Y%m%d%H%M%S |
TZ=UTC+0 utconv             | # UNIX 時間に変換
awk  '{print $1+86400}'     | # 有効期限を 1 日としてみた
TZ=UTC+0 utconv -r          | # UNIX 時間から逆変換
awk '{                      #   "Wdy, DD-Mon-YYYY HH:MM:SS GMT"形式に変換
  split("Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec",monthname);
  split("Sun Mon Tue Wed Thu Fri Sat",weekname);
  Y = substr($0, 1,4)*1; M = substr($0, 5,2)*1; D = substr($0, 7,2)*1;
  h = substr($0, 9,2)*1; m = substr($0,11,2)*1; s = substr($0,13,2)*1;
  Y2 = (M<3) ? Y-1 : Y; M2 = (M<3)? M+12 : M;
  w = (Y2+int(Y2/4)-int(Y2/100)+int(Y2/400)+int((M2*13+8)/5)+D)%7;
  printf("%s, %02d-%s-%04d %02d:%02d:%02d GMT¥n",
         weekname[w+1], D, monthname[M], Y, h, m, s);
}'
```

Cookie を手づくりするのも、本書のレシピをもってすれば十分可能だ。

参照

→レシピ 4.2（URL エンコードする）
→レシピ 3.3（シェルスクリプトで時間計算を一人前にこなす）
→レシピ 4.9（シェルスクリプトおばさんの手づくり Cookie(読み込み編)）

→ RFC 6265 文書[*23]
→ RFC 2616 文書[*24]

レシピ 4.11　シェルスクリプトによる HTTP セッション管理

問題

ショッピングカートを作りたい。そのためには買い物カゴを実装する必要があり、HTTP セッションが必要になる。どうすればよいか？

回答

mktemp コマンド[*25]を使って一時ファイルを作り、そこにセッション内で有効な情報を置くようにするのがよい。また mktemp で作った一時ファイルの名前はランダムなので、これをセッション ID に利用する。セッション ID はクライアント（Web ブラウザー）とやり取りする必要があるが、それには Cookie を利用すればよい。

次項でシェルスクリプトで HTTP セッションを管理するデモプログラムを紹介するが、セッションファイルを管理する部分をコマンド化したものも用意しているので、手っ取り早く済ませたい人は「解説」を参照されたい。

HTTP セッション実装の具体例

シェルスクリプトが自力で HTTP セッション管理を行うための要点をまとめたデモプログラムを紹介する。まず、このデモプログラム動作は次のとおりだ。

- 初めてアクセスすると、セッションが新規作成され、ウェルカムメッセージを表示する。
- 1 分以内にアクセスすると、セッションを延命し、前回アクセス日時を表示する。
- 1 分以降 2 分以内にアクセスすると、セッションは有効期限切れだが、Cookie によって以前にアクセスされたことを覚えているので「作り直しました」と表示する。
- 2 分以降経ってアクセスすると、以前にアクセスしたことを完全に忘れるので、新規の時と同じ動作をする。

[*23] http://tools.ietf.org/html/rfc6265
[*24] http://tools.ietf.org/html/rfc2616
[*25] mktemp コマンドは POSIX で規定されたコマンドではないのだが、多くの UNIX 系 OS に広く普及しているとして使わせてもらった。POSIX の範囲でも類似のものを作れなくはないが、/dev/urandom 等の良質な乱数源が必要になり、やはり POSIX を逸脱せざるを得ない。→レシピ 5.22（mktemp コマンド）参照

■HTTP セッション管理デモスクリプト

```
#! /bin/sh

# --- 0) 各種定義 ----------------------------------------------------
Dir_SESSION='/tmp/session'      # セッションファイル置き場
SESSION_LIFETIME=60             # セッションの有効期限 (1 分にしてみた)
COOKIE_LIFETIME=120             # Cookie の有効期限 (2 分にしてみた)

# --- 1)Cookie からセッション ID を読み取る ------------------------------
session_id=$(printf '%s' "${HTTP_COOKIE:-}"                        |
             sed 's/&/%26/g; s/[;, ]¥{1,¥}/¥&/g; s/^&//; s/&$//' |
             cgi-name                                              |
             nameread session_id                                   )

# --- 2) セッション ID の有効性検査 --------------------------------------
session_status='new'                          # デフォルトは「要新規作成」とする
while :; do
  # --- セッション ID 文字列が正しい書式 (英数字 16 文字とした) でないなら NG
  printf '%s' "$session_id" | grep -q '^[A-Za-z0-9]¥{16¥}$' || break
  # --- セッション ID 文字列で指定されたファイルが存在しないなら NG
  [ -f "$Dir_SESSION/$session_id" ] || break
  # --- ファイルが存在しても古すぎだったら NG
  touch -t $(date '+%Y%m%d%H%M%s'                 |
             utconv                               |
             awk "{print ¥$1-$SESSION_LIFETIME}" |
             utconv -r                            |
             awk 'sub(/..$/,".&")'                ) $Tmp-session_expire
  find "$Dir_SESSION" -name "$session_id" -newer $Tmp-session_expire | awk 'END{exit (NR!=0)}'
  [ $? -eq 0 ] || { session_status='expired'; break; }
  # --- これらの検査に全て合格したら使う
  session_status='exist'
  break
done

# --- 3) セッションファイルの確保 (あれば延命、なければ新規作成) -------
case $session_status in
  exist) File_session=$Dir_SESSION/$session_id
         touch "$File_session";;                              # セッションを延命する
  *)     File_session=$(mktemp $Dir_SESSION/XXXXXXXXXXXXXXXX)
         [ $? -eq 0 ] || { echo 'cannot create session file' 1>&2; exit; }
         session_id=${File_session##*/};;
esac

# --- 4)-1 セッションファイル読み込み ----------------------------------
msg=$(cat "$File_session")
case "${msg}${session_status}" in
  new)     msg="はじめまして! セッションを作りました。(ID=$session_id)";;
  expired) msg="セッションの有効期限が切れたので、作り直しました。(ID=$session_id)";;
esac

# --- 4)-2 セッションファイル書き込み ----------------------------------
printf ' 最終訪問日時は、%04d 年 %02d 月 %02d 日 %02d 時 %02d 分 %02d 秒です。(ID=%s)' ¥
       $(date '+%Y %m %d %H %M %S') "$session_id"                    ¥
       > "$File_session"                                                    (→次頁へ続く)
```

```
(→前頁からの続き)

# --- 5)Cookie を焼く ------------------------------------------------
cookie_str=$(echo "session_id ${session_id}"              |
             mkcookie -e+${COOKIE_LIFETIME} -p / -s Y -h Y)

# --- 6)HTTP レスポンス作成 ------------------------------------------
cat <<-HTTP_RESPONSE
    Content-type: text/plain; charset=utf-8$cookie_sid

    $msg
HTTP_RESPONSE
```

尚、このシェルスクリプトを動かすには、レシピ 3.3（シェルスクリプトで時間計算を一人前にこなす）で紹介した utconv コマンド、及びレシピ 4.10（シェルスクリプトおばさんの手づくり Cookie(書き込み編)）で紹介した mkcookie コマンドが必要になるので、実際に試してみたい人は予め準備しておくこと。

解説

「回答」で例示した HTTP セッション管理デモプログラムが行っている作業の流れは次のとおりである。

1) Web ブラウザーが申告してきたセッション ID があれば、それを Cookie から読む。
2) そのセッション ID が有効なものかどうか審査する。
3) 有効であればセッションファイルが存在するはずなのでタイムスタンプ更新をして延命し、無効であれば新規作成する。
4) セッションファイルの内容を見つつ、それに応じた応答メッセージを作成し、セッションファイルに書き込む。
5) 改めてセッションファイルを Web ブラウザーに通知するため、Cookie 文字列を作成する。
6) Cookie ヘッダーと共に応答メッセージを Web ブラウザーに送る。

今回は、手順 3) において有効セッションがあった場合は単に延命しただけであったが、セキュリティーを強化したいならここでセッション ID を付け替えてもよい。

しかしながら毎回いちいち書くにはちょっとコードが多いような気もする。厳密に言うと、**セッションファイルに書き込みを行う場合はファイルのロック（排他制御）も、これとは別に必要**なのである。そこで、せめて手順 2)、3) の処理を簡単に書けるよう、例によってコマンド化したものを用意した。HTTP セッションで用いるファイルの管理用コマンドということで “sessionf” である[*26]。mktemp という非 POSIX コマンドを使っているために POSIX 完全準拠とはいかないのだが。

sessionf コマンドの使い方

まず前述のスクリプトの置き替えで使用例を示す。つまり、有効なセッションが無ければ新規作成し、有れば延命するというパターンだ。それには、デモスクリプトの 2)〜3) の部分を

```
File_session=$(sessionf avail "$session_id" at=$Dir_SESSION/XXXXXXXXXXXXXXXXXXXXXXXX" ¥
                                            lifemin=$SESSION_LIFETIME                 )
case $? in 0) session_status='exist';; *) session_status='new';; esac
session_id=${File_session##*/}
```

[*26] https://github.com/ShellShoccar-jpn/misc-tools/blob/master/sessionf にアクセスし、そこにあるソースコードをコピー&ペーストしてもよいし、あるいは “RAW” と書かれているリンク先を「名前を付けて保存」してもよい。

と、書き換えればよい。たったの 4 行になる。sessionf のサブコマンド “avail” は、有効なものがあれば延命、無ければ新規作成を意味する。そして後ろの “at” プロパティーは、セッションファイルの場所と、無かった場合のセッションファイルのテンプレートを mktemp と同じ書式で指定するものであり、“lifetime” プロパティーは、有効期限を判定するための秒数を指定するものだ。

一方、セキュリティーを高めるため、有効なセッションがあった場合には既存のセッションファイルの名前と共にセッション ID を付け替えたい、ということであればサブコマンドを “renew” にするだでよい。

```
File_session=$(sessionf renew "$session_id" at=$Dir_SESSION/XXXXXXXXXXXXXXXXXXXXXXXX" ¥
                                          lifemin=$SESSION_LIFETIME                  )
case $? in 0) session_status='exist';; *) session_status='new';; esac
session_id=${File_session##*/}
```

sessionf の詳しい使い方については、ソースコードの冒頭にあるコメントを参照されたい。

“XXXX...” は長めにするべき

これは、sessionf コマンドというより、依存している mktemp コマンドに起因する問題であるが、ランダムな文字列の長さを指定するための “XXXX...” という記述の “X” は長めにするべきである。理由は、CentOS 5 を動かせる環境があれば実際に試してみるとよくわかる。

■CentOS 5 で mktemp コマンドを実行すると……

```
$ mktemp /tmp/XXXXXXXX ↵
/tmp/OyA10700
$ mktemp /tmp/XXXXXXXX ↵
/tmp/sPr10701
$
```

生成されたランダムなはずのファイル名の末尾を見ると数字になっている。なんとこれはその時に発行されたプロセス ID なのだ。つまり、**CentOS 5 の mktemp コマンド実装は、ランダム文字列としての質が低いということだ。**だから文字列を長くしてランダム文字列の不規則性を高めてやらなければならない。ちなみに CentOS 6 以降ではこの問題は解消されている。

参照

→レシピ 3.3（シェルスクリプトで時間計算を一人前にこなす）
→レシピ 4.9（シェルスクリプトおばさんの手づくり Cookie(読み込み編)）
→レシピ 4.10（シェルスクリプトおばさんの手づくり Cookie(書き込み編)）
→レシピ 5.22（mktemp コマンド）

第 5 章

どの環境でも使えるシェルスクリプトを書く

シェルスクリプトは環境依存が激しいから……

などとよく言われ、敬遠される。それなら共通しているものだけ使えばいいのだが、それについてまとめているところがなかなかないので書いてみることにした。

「どの環境でも使える」ようにするには？

まずは定義から

まずは、何をもって「どの環境でも使える」とするのかについて定義する。じつはこれがなかなか難しい。あまりこだわりすぎると「古いものも含め、既存の UNIX 全てで使えるものでなければダメ」ということになってしまう。しかし、私個人としては**今も現役（＝メンテナンスされている）の UNIX 系 OS で使いまわせること**にこだわりたい。そこで

「どの環境でも使える」＝「POSIX で規定されている」

と定義することにした。

とはいっても全ての OS やディストリビューションについて調べられるわけではないので、この記事では基本的に最新の POSIX[*1]で定義されていることをもって、どの環境でも使えると判断するようにした。従って、互換性確保のため、シェルの中で使ってよい機能は **Bourne シェルの範囲**ということにし、bash,ksh,zsh, あるいは csh 等の拡張機能は使わないようにする。

ただし、いくら POSIX で規定されているといっても実際の環境でそれを採用しているものが稀であるとか、POSIX の範囲ではどうしても不足していてるものの殆どの環境で同じように使える、といったものに関しては書くことにする。つまり「基本的には POSIX に準拠する」ということとし、実用性と乖離したドキュメントにするつもりはない。

結局どうすればいいのか

というわけで、「どの環境でも使える」シェルスクリプトを書くのであれば基本的に POSIX の範囲で書くように気を付けることだ。具体的には “IEEE Std 1003.1” を記した Web ページ[*2]を確認、特に “Shell & Utilities” というメニューに書いてある文法やコマンド仕様を読み、今から使おうとしている文法・コマンド・コマンド引

[*1] 執筆時点の最新は “IEEE Std 1003.1, 2013 Edition” である

[*2] “ieee” と “POSIX” という単語で検索すれば辿り着く。執筆時は 2013 年版が最新で URL は `http://pubs.opengroup.org/onlinepubs/9699919799/`である。

数がそこで規定されているかどうかを確認する。

併せて、本章に記したレシピを頭に入れておくといいだろう。POSIX ドキュメントに基づく解説のみならず、現場で得た実戦的なノウハウも記してある。

■第一部 文法・変数など

まずはシェル自身の文法や変数、多くの UNIX コマンドに共通する仕様などについて、どの環境でも動くようにするための注意点を記す。

レシピ 5.1　シェル変数

まず配列は使えない。従って bash に存在する組込変数である `PIPESTATUS` も使えない。レシピ 3.1（PIPESTATUS さようなら）を参照してもらいたい。

変数の中身を部分的に取り出す記述に関して使っても大丈夫なものに関しては、POSIX の第 2.6.2 項[*3]を見るとまとまっている。

レシピ 5.2　スコープ

→レシピ 5.10（local 修飾子）を参照

レシピ 5.3　正規表現

これは AWK、grep、sed 等、コマンドによっても使えるメタキャラは違うし、`grep` なら ‘-E‘オプションを付けるかどうかでも違うし、さらに GNU 版でしか使えないものもあるので注意が必要。`grep` コマンド*BSD 上でも GNU 版が採用されている場合がある。→レシピ 5.18（grep コマンド）参照

しかし、 **正規表現メモ**[*4]というスバラシいまとめページがあるのでここを見れば、使っても互換性が維持できるメタキャラがすぐわかる。

え、シェル変数の正規表現？それは一部シェルの独自拡張機能なので使えない。

→レシピ 5.6（ロケール）、レシピ 5.4（文字クラス）も参照

レシピ 5.4　文字クラス

`[[:alnum:]]` のように記述して使う「文字クラス」というものがある。だが、文字クラスは使わない方が無難だ。

これの正式名称は「POSIX 文字クラス」[*5]という。その名のとおり POSIX 準拠であるのだが、Raspberry Pi の AWK など、一部の実装ではうまく動いてくれない。

まぁ、POSIX に準拠してないそっちの実装が悪いといってしまえばそれまでなのだが、そもそも設定されているロケールによって全角を受け付けたり受け付けなかったりして環境の影響を受けやすいので使わない方がよいだろう。

[*3] `http://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html#tag_18_06_02`

[*4] `http://www.kt.rim.or.jp/~kbk/regex/regex.html`

[*5] 使えるもの一覧は、`http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html#tag_09_03_05` 参照

レシピ 5.5　乱数

乱数を求めたい時、bash の組込変数 RANDOM を使うのは論外だが、「それなら」と AWK コマンドの rand 関数と srand 関数を使えばいいやと思うかもしれないがちょっと待った！

論より証拠。FreeBSD で次の記述を何度も実行してみれば、非実用的であることがすぐわかる。

```
$ for n in 1 2 3 4 5; do awk 'BEGIN{srand();print rand();}'; sleep 1; done ↵
0.0205896
0.0205974
0.0206052
0.020613
0.0206209
$
```

つまり動作環境によっては乱数としての質が非常に悪いのだ。AWK が内部で利用している OS 提供ライブラリ関数の rand() と srand() を、FreeBSD は低品質だったオリジナルのまま残し、新たに random() という別の高品質乱数源関数を提供することで対応しているのが理由なのだが……。（Linux では rand() と srand() を内部的に random() にしている）

/dev/urandom を使うのが現実的

ではどうすればいいか。POSIX で定義されているものではないが、/dev/urandom を乱数源に使うのが現実的だと思う。例えば次のようにして dd、od、sed コマンドを組み合わせれば 0〜4294967295 の範囲の乱数が得られる。

```
$ dd if=/dev/urandom bs=1 count=4 2>/dev/null | od -A n -t u4 | sed 's/[^0-9]//g' ↵
```

最後の段で tr コマンドではなく sed コマンドを使っている理由については、レシピ 5.30（tr コマンド）を参照。

レシピ 5.6　ロケール

どの環境でも動くことを重視するなら、環境変数の中でもとりわけロケール系環境変数の内容には注意しなければならない。理由は、ロケール環境変数（LANG や LC_*）の内容によって動作が変わるコマンドがあるからだ。

具体的に何が変わるかといえば、主に文字列長の解釈や、出力される日付である。下記にそれらをまとめてみた。

ロケール系環境変数の影響を受けるもの

入力文字列の解釈が変わるもの

例えば環境変数 LANG や LC_*等の内容によって、全角文字を半角の相当文字と同一扱いしたり、全角文字の文字列長を 1 とするものとして、次のようなものがある。

- AWK コマンド、grep コマンド、sed コマンド等の正規表現（[[:alnum:]]、[[:blank:]] 等の文字クラスや、+、¥{n,m¥}などの文字数指定子）
- AWK コマンドの文字列操作関数（length、substr）
- wc コマンドの文字数（-m オプション）

など。

列区切り文字が変わるもの

環境変数 LANG の内容によって、デフォルトの列区切り文字に全角スペースが加わるもの。

- join コマンド、sort コマンド等（-t オプション）

など。

出力フォーマットが変わるもの

環境変数（LANG や LC_*）の内容によって、出力される文字列や書式が変わるもの。

- date コマンドのデフォルト日時フォーマット
- df コマンドの 1 行目の列名の言語
- ls コマンド-l オプションのタイムスタンプフォーマット
- シェルの各種エラーメッセージ

など。

通貨や数値のフォーマットが変わるのも

環境変数 LC_MONETARY や LC_NUMERIC の影響を受けるもの。

- sort……-n オプションを指定した場合に、桁区切りのカンマの影響を受けたり受けなかったりする。

対策

全ての環境で動くようにするのであれば、ロケール設定無しの状態、すなわち英語で使うべきであろう。対策方法を 3 つ紹介する。

■env コマンドで全環境変数を無効化してコマンド実行

```
echo ’ほげ HOGE’ | env -i awk ’{print length($0)}’
```

■LC_ALL=C を設定し、C ロケールにしてコマンド実行

```
echo ’ほげ HOGE’ | LC_ALL=C awk ’{print length($0)}’
```

■予め LC_ALL=C を設定しておく

```
export LC_ALL=C                                  # シェルスクリプトの冒頭でこれを実行
   :
   :
echo ’ほげ HOGE’ | awk ’{print length($0)}’ # そして目的のコマンドを実行
```

ちなみに、いにしえの export は、=を使って変数の定義と export 化を同時に行えないということだが、今ど

きの POSIX の man ページ[*6]によれば使えることになっている。

レシピ 5.7　$((式))

よく「expr コマンドを使え」というが、今どきは`$((式))` も POSIX で規定されており、使っても問題無い。

ただ、数字の頭に “0” や “0x” を付けると、それぞれ 8 進数、16 進数扱いされるので expr コマンドとの間で移植をする場合は気を付けなければならない。（expr コマンドは、数字の先頭に “0” が付いていても常に 10 進数と解釈される）

```
$ echo $((10+10)) ↵   ← 10 進数の 10 に、10 進数の 10 を足す
20
$ echo $((10+010)) ↵  ← 10 進数の 10 に、8 進数の 10 を足す
18
$ echo $((10+0x10)) ↵ ← 10 進数の 10 に、16 進数の 10 を足す
26
$
```

この問題は、異なる実装の AWK 間にもあるので注意。→レシピ 5.13（AWK コマンド）参照

レシピ 5.8　case 文

→レシピ 5.9（if 文）参照

レシピ 5.9　if 文

たまに、else の時は何かしたいけど then の時は何もしたくないということがある。だからといって then と else の間に何も書かないと、bash 等一部のシェルではエラーを起こしてしまう。

■bash の場合、次のコードはエラーになる

```
if [ -s /tmp/hoge.txt ]; then
  # 1 バイトでも中身があれば何もしない ←ここでエラー
else
  # 0 バイトだったら消す
  rm /tmp/hoge.txt
fi
```

elif の後も else の後も同様であるし、case 文でも条件分岐した先に何もコードを書いていなければ同じだ。要するに **bash では、条件分岐先に有効なコードを置かないというコードが許されない**のだ。（コメントを書いただけではダメ）

対策

何らかの無害な処理を書けばいいのだが、一番軽いのは null コマンド（“:”）ではないだろうか。つまり、こう書けばどの環境でも無難に動くようになる。

[*6] http://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html#export

■何もしたくなければ null コマンドを置くとよい（3 行目に注目）

```
if [ -s /tmp/hoge.txt ]; then
  # 1 バイトでも中身があれば何もしない ←今度は bash でもエラーにならない
  :
else
  # 0 バイトだったら消す
  rm /tmp/hoge.txt
fi
```

別の対策としては、条件を反転してそもそも else 節を使わずに済むようにするのもいいだろう。しかしそれによってコードが読みにくくなったり、条件が 3 つ以上の複雑な場合などは、無理せずこの技法を用いるべきだ。

レシピ 5.10　local 修飾子

シェル関数の中で用いる変数を、その関数内だけで有効なローカル変数にする場合に用いる修飾子だが、これは POSIX では規定されていない。しかし、関数内ローカルな変数は簡単に用意できる。小括弧で囲ってサブシェルを作ればその中で代入した値は外へは影響しないからだ。

次のシェル関数 “localvar_sample()” を見てもらいたい。中身を丸ごと小括弧で囲ったシェル関数で定義した次のシェル変数$a、$b、$c は、関数終了後に消滅するし、外部に同名の変数があってもその値を壊すことはない。（だだし初期値はそちらの値になっている）

■シェル関数内でローカルな変数を作る

```
localvar_sample() {
  (                   # ←小括弧で囲む
    a=$(whoami)
    b='My name is'
    c=$(awk -v id=$a -F : '$1==id{print $5}' /etc/passwd)
    echo "$b $c."
  )
}
```

レシピ 5.11　PIPESTATUS 変数

例えば組込変数 PIPESTATUS に依存したシェルスクリプトが既にあって、それをどの環境でも使えるように書き直したいと思った場合、実は可能だ。詳しいやり方については、レシピ 3.1（PIPESTATUS さようなら）を参照してもらいたい。

■第二部 コマンド

いくらシェルスクリプトの文法に気をつけても、呼び出すコマンドが一部の環境でしか通用しないような使い方では意味をなさない。次に、どの環境でも使えるシェルスクリプトを書くために気をつけるべきコマンドの各論を紹介する。

レシピ 5.12　“[” コマンド

→レシピ 5.29（test コマンド）参照

レシピ 5.13　AWK コマンド

AWK はそれが 1 つの言語でもあるので、説明しておくべきことがたくさんある。

-0（マイナス・ゼロ）

FreeBSD 9.x に標準で入っている AWK では、-1*0 を計算すると “-0” という結果になる。

■FreeBSD 9.1 で $-1 * 0$ を計算させてみると

```
$ awk 'BEGIN{print -1*0}' ↵
-0
$
```

ところがこの挙動は同じ FreeBSD でも 10.x では確認されないし、GNU 版 AWK でも起こらないようだ。

このようにして、同じ 0 であっても “-0” という二文字で返してくる場合のある実装もあるので注意してもらいたい。

0 始まり即値の解釈の違い

頭に 0 が付いている数値を即値（プログラムに直接書き入れる値）として与えると、それを 8 進数と解釈する AWK 実装もあれば 10 進数と解釈する AWK 実装もある。

■FreeBSD の AWK で即値の 010 を解釈させた場合

```
$ awk 'BEGIN{print 010;}' ↵
10
$
```

■GNU 版 AWK で即値の 010 を解釈させた場合

```
$ awk 'BEGIN{print 010;}' ↵
8
$
```

どこでも同じ動きにしたければ文字列として渡せばよい。すると 10 進数扱いになる。

■GNU 版 AWK でも文字列として"010"を渡せば 10 進数扱いされる

```
$ awk 'BEGIN{print "010"*1;}' ↵
10
$ echo 010 | awk 'BEGIN{print $1*1;}' ↵
10
$
```

length 関数の機能制限

大抵の AWK 実装は、

```
$ awk 'BEGIN{split("a b c",chr); print length(chr);}' ↵
3
$
```

とやると、きちんと要素数を返すだろう。しかし実装によってはこれに対応しておらず、エラー終了してしまうものがある。このため、例えば次ようにユーザー関数 `arlen()` を作り、配列の要素数はその関数で数えるようにすべきだ。

■配列の要素数を数える関数を自作しておく

```
awk '
  BEGIN{split("a b c",chr); print arlen(chr);}
  function arlen(ar,i,l){for(i in ar){l++;}return l;}
'
```

幸い、**AWK の配列変数は参照渡し**なので要素の中身が膨大だとしてもそれは影響しない。（要素数が大きい場合はやはり負担がかかると思うのだが……）

length() が使えるなら使いたい

「length 関数が使えるなら使いたい！」というワガママなアナタは、こうすればいい。

■length が使えるなら使いたいワガママなアナタへ

```
# シェルスクリプトの冒頭で、配列に対して length() を使ってもエラーにならないことを確認
if awk 'BEGIN{a[1]=1;b=length(a)}' 2>/dev/null; then
  arlen='length'  # ←エラーにならないなら length
else
  arlen='arlen'   # ←エラーになるなら独自関数"arlen"
fi

awk '
  BEGIN{split("a b c",chr); print '$arlen'(chr);}       # ←判定結果に応じて適宜選択される
  function arlen(ar,i,l){for(i in ar){l++;}return l;}
'
```

printf、sprintf 関数

→レシピ 5.24（printf コマンド）参照

rand 関数,srand 関数は使うべきではない

→レシピ 5.5（乱数）参照

gensub 関数は使えない

GNU 版 AWK には独自拡張がいくつかあるが、中でも注意すべき点は gensub 関数がそれであること。互換性を優先するなら、多少不便かもしれないが sub 関数や gsub 関数を使こと。その他、こまごまと気を付けるべきことについては「GNU AWK の ‘–posix‘オプションに関するまとめ@kbk さんの Web ページ[*7]が大変参考になる。

正規表現では有限複数個の繰り返し指定ができない

AWK の正規表現は繰り返し指定が苦手。文字数指定子のうち、“?”（0〜1 個）と “*”（0 個以上）と “+”（1 個以上）は使えるが、2 個以上の任意の数を指定するための “{数}” には対応していない。GNU 版 AWK では独自拡張して使えるようになっているのだが。

その他の基本正規表現[*8]については全部使えるのだが、正規表現メモさん Web サイトの AWK の記述に[*9]に詳しくまとまっているので、そちらを見るのが便利だろう。

整数の範囲

例えば、あなたの環境の AWK は次のように表示されはしないだろうか？

```
$ awk 'BEGIN{print 2147483648}' ↵
2.14748e+09
```

上記の例は、0x7FFFFFFF($= 2^{32} - 1$) より大きい整数を扱えない AWK 実装である。このようなことがあるので、桁数の大きな数字を扱わせようとする時は注意が必要だ。計算をせず、単に表示させたいだけなら文字列として扱えばよい。

ロケール

→レシピ 5.6（ロケール）を参照

レシピ 5.14　date コマンド

元々の機能が物足りないがゆえか、各環境で独自拡張されているコマンドの一つだ。だが互換性を考えるなら、使えるのは

- -u オプション（=UTC 日時で表示）

[*7] http://www.kt.rim.or.jp/~kbk/gawk-30/gawk_15.html#SEC135

[*8] POSIX ドキュメントの “9.3 BRE”（http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html#tag_09_03）において規定されているメタ文字

[*9] http://www.kt.rim.or.jp/~kbk/regex/regex.html#AWK

- “+ フォーマット文字列” にて表示形式を指定

の 2 つだけと考えるが無難だろう。尚、フォーマット文字列中に指定できるマクロ文字の一覧は、POSIX の date コマンドの man ページ[*10]の “Conversion Specifications” の段落にまとめられているので参照されたい。

UNIX 時間との相互変換

マクロの種類はいろいろあるのだが、残念ながら UNIX 時間[*11]との相互変換は無い。これさえできれば何とでもなるのだが……。

しかしこんなこともあろうかと、相互変換を行うコマンドを作ったのだ。もちろんシェルスクリプト製である。詳しくは、レシピ 3.3（シェルスクリプトで時間計算を一人前にこなす）を参照してもらいたい。

レシピ 5.15　du コマンド

特定ディレクトリー以下のデータサイズを求めるこのコマンド、POSIX で規定されているオプションではないが-h というものがある。これはファイルやディレクトリーのデータサイズを k（キロ）、M（メガ）、G（ギガ）等最適な単位を選択して表示するものだ。

しかしこのオプションの表示フォーマットは、環境によって僅かに異なる。

■FreeBSD の du コマンド-h オプションの挙動

```
$ du -h /etc | head -n 10 ↵
118K    /etc/defaults
2.0K    /etc/X11
372K    /etc/rc.d
4.0K    /etc/gnats
6.0K    /etc/gss
 30K    /etc/security
 40K    /etc/pam.d
4.0K    /etc/ppp
2.0K    /etc/skel
144K    /etc/ssh
$
```

■Linux の du コマンド-h オプションの挙動

[*10] http://pubs.opengroup.org/onlinepubs/9699919799/utilities/date.html
[*11] エポック秒とも呼ばれる “UTC 1970/1/1 00:00:00” からの秒数

```
$ du -h /etc | head -n 10 ↵
112K    /etc/bash_completion.d
12K     /etc/abrt/plugins
4.0K    /etc/statetab.d
4.0K    /etc/dracut.conf.d
28K     /etc/cron.daily
4.0K    /etc/audisp
4.0K    /etc/udev/makedev.d
36K     /etc/udev/rules.d
48K     /etc/udev
8.0K    /etc/sasl2
$
```

違いがわかるだろうか？ 1 列目（サイズ）が、前者は右揃えなのに後者は左揃えなのだ。従ってどちらの環境でも動くようにするには、1 列目であっても行頭にスペースが入る可能性を考慮しなければならない。

例えば 1 列目の最後に単位"B"を付加したいとしたら、下記の 1 行目はダメで、2 行目の記述が正しい。

■1 行目の最後に"B"(単位) を付けたい場合

```
du -h /etc | sed 's/^[0-9.]¥{1,¥}[kA-Z]/&B/'   # ←これでは不完全

du -h /etc | sed 's/^ *[0-9.]¥{1,¥}[kA-Z]/&B/' # ←こうするのが正しい

du -h /etc | awk '{$1=$1 "B";print}'           # ←折角の桁揃えがなくなるがまぁアリ
```

このようにして 1 列目にインデントが入るコマンドは結構あるし、インデントの幅も環境によりまちまちなので注意が必要だ。（例、 `uniq -c` 、`wc` などなど）

レシピ 5.16　echo コマンド

例えば次のシェルスクリプト “`echo_test.sh`” を見てもらいたい。これは引数で与えられた文字列を 1 行ずつ表示するという動きをするように作ってある。

■引数を 1 つ 1 行で表示するシェルスクリプト echo_test.sh

```
#! /bin/sh
for arg in "$@"; do
  echo "$arg"
done
```

このシェルスクリプトで、引数の一つに “`-e`” を付けて実行してみる。FreeBSD では`-e` もちゃんと表示される一方で、例えば`/bin/sh` の正体が bash になっている Linux の場合は “`-e`” がうまく表示されない。理由は、bash の echo コマンドが “`-e`” を、表示すべき文字列ではなくてオプションとして解釈するからだ。

■FreeBSD の sh で前述のシェルスクリプトを動かすと……

```
$ ./echo_test.sh -s -e -d ↵
-s
-e
-d
$
```

■Linux の bash で前述のシェルスクリプトを動かすと……

```
$ ./echo_test.sh -s -e -d ↵
-s

-d
$
```

これは bash で実装されている echo コマンドは、“-e” を文字列ではなくオプションとして解釈するためだ。つまり上記のコードは環境によって挙動が変わっているわけで、互換性に問題があるということになる。

ちなみに POSIX における sh の man ページ[*12]によれば、**echo にはオプションが全く規定されていない**[*13]。従って、どの環境でも動くシェルスクリプトを目指すなら、FreeBSD の sh でも使える-n オプションすら使うべきではないのだ。

対策

このように、環境によっては与えられた文字列がオプションとみなされて意図せぬ動作をするので、**どんな文字列が入っているかわからない変数を扱いたければ printf コマンド**を使うようにすべきだ。

■echo のオプション反応問題を回避する対策を講じたもの

```
#! /bin/sh
for arg in "$@"; do
  printf '%s¥n' "$arg"
done
```

もちろん、ハイフンで始まらないとわかっているならそのままでよいのだが。

レシピ 5.17　exec コマンド

注意すべきは exec コマンド経由で呼び出すコマンドに環境変数を渡したい時だ。

例えば、exec コマンドを経由しない場合、コマンドの直前で環境変数を設定し、コマンドに渡すことができる。

[*12] http://pubs.opengroup.org/onlinepubs/9699919799/utilities/echo.html

[*13] 詳しく読むと「System V など-n オプションが効かない環境があるのでそれがやりたい時は printf コマンドを使え」と書いてある。

```
$ name=val awk 'BEGIN{print ENVIRON["name"];}' ↵
val
$
```

しかし、exec コマンドを環境変数の直後に挿むと、何も表示されないシェルがある。

```
$ name=val exec awk 'BEGIN{print ENVIRON["name"];}' ↵

$
```

一部の環境の exec コマンドは、このようにして設定された環境変数を渡してくれないからだ。
もし exec コマンド越しに環境変数を渡したいのであれば、事前に export で設定しておくこと。

```
$ export name=val ↵
$ awk 'BEGIN{print ENVIRON["name"];}' ↵
val
$
```

あるいは、exec の後に env コマンドを経由させるのでもよい。

```
$ exec env name=val awk 'BEGIN{print ENVIRON["name"];}' ↵
val
$
```

レシピ 5.18　grep コマンド

俺は*BSD を使っているから、grep だって GNU 拡張されていない BSD 版のはず。ここで使えるメタ文字はどこでも使えるでしょ。

と思っているアナタ。果たして本当にそうか確認してみてもらいたい。

■アナタの grep はホントに BSD 版？

```
$ grep --version ↵
grep (GNU grep) 2.5.1-FreeBSD ↵

Copyright 1988,1992-1999,2000,2001 Free Software Foundation,Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

$
```

なんと、GPL ソフトウェア排除に力を入れている FreeBSD でも、grep コマンドは GNU 版だ。関係者によれば、主に速さが理由で、grep だけは当面 GNU 版を提供するのだという。よって、POSIX 標準だと思っていたメタ文字が実は GNU 拡張だったということがある。代表的なものは “¥+” や “?” や “¥|” である。

POSIX 標準 grep で使える正規表現は、-E オプション無しの場合には POSIX の 9.3 節 “Basic Regular Expression”[*14]で規定されているものだけ。-E オプション付きの場合には同 9.4 節 “Extended Regular Expression”[*15]で規定されているものだけだ。詳しくは、「正規表現メモ」さんによる日本語解説[*16]が分かりやすいかもしれない。

レシピ 5.19　head コマンド

大抵の環境の head コマンドは、-c オプション（ファイルの先頭をバイト単位で切り出す）に対応している。しかし実は、**POSIX では head コマンドに-c オプションは規定されていない。**現に、正しく実装されていない環境も存在する[*17]。

ちなみに、POSIX でも tail コマンドでは-c オプションがきちんと規定されているので、head にだけ規定されていないのはちょっと不思議だ。

対策

さて、それでは-c オプションが使えない環境で何とかして同等のことができないものか……。大丈夫、dd コマンドでできる。

試に “12345” という 5 バイト（改行コードを加えれば 6 バイト）の文字列から先頭の 3 バイトを切り出してみよう。bs（ブロックサイズ）を 1 バイトとして、それを 3 つ（count）と指定すればよい。

```
$ echo 12345 | dd bs=1 count=3 2>/dev/null ↵
123$
```

これは標準入力のデータを切り出す例だったが、if キーワードを使えば実ファイルでもできる。

```
echo 12345 >/tmp/hoge.txt ↵
$ dd if=/tmp/hoge.txt bs=1 count=3 2>/dev/null ↵
123$
```

尚、dd コマンドは標準エラー出力に動作結果ログを吐くので、head -c 相当にするなら dd コマンドの最後に 2>/dev/null などと書いて、ログを捨てること。

レシピ 5.20　ifconfig コマンド

これも POSIX で規定されていないコマンドだし、最近では Linux など使わない傾向にあるコマンドであるが、全ての環境で動くことを目指すならまだまだ外せないコマンドである。

[*14] http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html#tag_09_03
[*15] http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html#tag_09_04
[*16] http://www.kt.rim.or.jp/~kbk/regex/regex.html#POSIX
[*17] AIX では最後に余計な改行コードが付く。

さて、実行中のホストに振られている IP アドレスを調べたい時にこのコマンドを使いたいことがあるが、各環境での互換性を確保するには 2 つのことに注意しなければならない。

パスが通っているとは限らない

大抵の場合、ifconfig は/sbin の中にある。しかし**多くの Linux のディストリビューションでは一般ユーザーに sbin 系のパスが通されていない**。だから、このコマンドを互換性を確保しつつ使いたい場合は、環境変数 PATH に sbin 系ディレクトリー（/sbin、/usr/sbin）を追加しておく必要がある。

フォーマットがバラバラ

ifconfig から返される書式が環境によってバラバラである。そこで、IP アドレスを取得するためのレシピを用意したので参照されたい。→レシピ 1.7（IP アドレスを調べる (IPv6 も)）参照

レシピ 5.21　kill コマンド

kill コマンドで送信シグナルを指定する際は、名称でも番号でも指定できるわけだが、番号で指定する場合は気を付けなければならない。POSIX の kill コマンドの man ページ[*18]によれば、どの環境でも使える番号は 5.1 に記したもの以外保証されていない。

表 5.1 POSIX で番号が約束されているシグナル一覧

Signal No.	Signal Name
0	0
1	SIGHUP
2	SIGINT
3	SIGQUIT
6	SIGABRT
9	SIGKILL
14	SIGALRM
15	SIGTERM

「え、たったこれだけ!?」と思うだろうか。もちろんシグナルの種類がこれだけしかないわけではない。ただ、**その他のシグナルは名称と番号が環境によってまちまち**なのだ。例えば “SIGBUS” は、FreeBSD では 10 だが、Linux では 7、といった具合である。

従って、上記以外のシグナルを指定したい場合は名称（”SIG”の接頭辞を略した文字列）で行うこと。使える名称自体は、POSIX でも規定されているとおり[*19]、豊富にある。

[*18] http://pubs.opengroup.org/onlinepubs/9699919799/utilities/kill.html
[*19] http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/signal.h.html

-l オプションは避ける

kill コマンドで-l オプションを指定すれば、使えるシグナルの種類の一覧が表示されるのはご存知のとおり。しかし、番号と名称の対応がこれで調べられるわけではない。Linux だと丁寧に番号まで表示されるが、FreeBSD では単に名称一覧しか表示されない（一応順番と番号は一致してはいるのだが）。

レシピ 5.22　mktemp コマンド

mktemp コマンドもやはり POSIX で規定されたものではない。よって、実際に使えない環境がある。

しかしシェルスクリプトを本気で使いこなすにはテンポラリーファイルが欠かせず、そんな時に便利なコマンドが mktemp なのだが……。どうすればいいだろうか。

一意性のみでセキュリティーは保証しない簡易的なもの[*20]なら、下記のようなコードを追加しておけば作れる。

■mktemp コマンドが無い環境で、その「簡易版」を用意するコード

```
which mktemp >/dev/null 2>&1 || {
  mktemp_fileno=0
  mktemp() {
    (
      filename="/tmp/${0##*/}.$$.$mktemp_fileno"
      touch "$filename"
      chmod "$filename"
      echo "$filename"
    )
    mktemp_fileno=$((mktemp_fileno+1))
  }
}
```

簡単に解説しておこう。最初に mktemp コマンドの有無を確認し、無ければコマンドと同じ使い方ができるシェル関数を定義するものだ。

ただし引数は無視され、必ず/tmp ディレクトリーに生成されるので、それでは都合が悪い場合は適宜書き換えておくこと。それから、“mktemp_fileno” という変数をグローバルで利用しているので書き換えないようにも注意すること。

レシピ 5.23　nl コマンド

POSIX でも規定されている-w オプションであるが、環境によって挙動が異なるので注意。（尚、-w オプションは POSIX でデフォルト値が設定されているため、**このオプションを記述しなくても同様の問題が起こるので注意！**[*21]）

-w オプションとは行番号に割り当てる桁数を指定するものであるが、問題は指定した桁数よりも桁があふれてしまった時である。溢れた場合の規定は定義されていないので、実装によって解釈が異ってしまったようだ。

2 つの実装を例にとるが、まず BSD 版の nl コマンドでは、溢れた分の上位桁は消されてしまう。

[*20] もしセキュリティーを確保したい場合は良質な乱数源が必要となり、そうなると/dev/urandom 等に頼らざるを得ない。→レシピ 5.5（乱数）参照

[*21] 一方、cat コマンドの-n オプションではこの問題は起こらないようだ。

■BSD 版 nl コマンドの場合

```
$ yes | head -n 11 | nl -w 1 ↵
1       y
2       y
3       y
4       y
5       y
6       y
7       y
8       y
9       y
0       y
1       y
$
```

一方、GNU 版の nl コマンドでは、溢れたとしても消しはせず、全桁を表示する。

■GNU 版 nl コマンドの場合

```
$ yes | head -n 11 | nl -w 1 ↵
1       y
2       y
3       y
4       y
5       y
6       y
7       y
8       y
9       y
10      y
11      y
$
```

行番号数字の直後につくのはデフォルトではタブ（“¥t”）なので、GNU 版では桁数が増えるとやがてズレることになる。BSD 版はズレることはない代わりに上位桁が見えないので、何行目なのかが正確にはわからない。

レシピ 5.24　printf コマンド

互換性を重視するなら、¥xHH（“HH” は任意の 16 進数）という 16 進数表記によるキャラクターコード指定をしてはいけない。これは一部の printf の独自拡張だからだ。代わりに¥000（“000” は任意の 8 進数）という 3

桁の 8 進数表記を用いること。

これは、AWK コマンドの printf 関数、sprintf 関数についても同様である。

レシピ 5.25　ps コマンド

現在の ps コマンドは、オプションにハイフンを付けない BSD スタイルなど、いくつかの流派が混ざっているので厄介だ。

-x オプションは避ける

「制御端末を持たないプロセスを含める」という働きであるが、このオプションは使わない方がいい。そもそも POSIX における ps コマンドの man ページ[*22]にはないし、少なくとも GNU 版と BSD 版では解釈が異なるようだ。

例えば CGI(httpd) によって起動されたプロセス上で、-a オプションも-x オプションも付けずに自分に関するプロセスのみを表示しようとした場合、前者では表示されるものが後者では-x を付けた場合に初めて表示されるなどの違いがある。

結局のところ、互換性を重視するなら、大文字である-A オプションを用いてとにかく全てを表示（-ax に相当）させる方がよいだろう。

-l オプションも避ける

-l オプションは、ls コマンドの同名オプションのように多くの情報を表示するためのものである。これは POSIX の ps コマンド man ページにも記載されているし、実際主要な環境でサポートされているので問題なさそうだが、使うべきではない。理由は、表示される項目や順序が OS やディストリビューションによってバラバラだからだ。

-o オプションほぼ必須

-l オプションを付けた場合の表示項目や順序がバラバラだと言ったが、実は付けない場合もバラバラだ。どの環境でも期待できる表示内容といえば、

- 1 列目に PID が来ること
- 行のどこかにコマンド名が含まれていること

くらいなものだ。互換性を維持しながらそれ以上の情報を取得しようとするなら、-o オプションを使って明確に表示させたい項目と順序を指定しなければならない。

-o オプションで指定できる項目一覧については POSIX の ps コマンド man ページ内の「STDOUT セクション」後半に記されている。（太小文字で列挙されている項目で、現在のところ”ruser”から”args”までが記されている）

[*22] http://pubs.opengroup.org/onlinepubs/9699919799/utilities/ps.html

補足. 親プロセス ID(PPID)

Linux では、親プロセス ID が 0 になるのは PID が 1 の “init” だけだ。しかし、FreeBSD 等では他の様々なシステムプロセスの場合はそれ以外のプロセスの親も 0 になる場合がある。これは、ps コマンドの違いというよりカーネルの違いであるが、互換性のあるプログラムを書くときには注意すべきところだ。

レシピ 5.26　sed コマンド

sed にもまた AWK 同様に、複数の注意すべき点がある。

最終行が改行コードでないテキストの扱い

試しに `printf 'Hello,¥nworld!' | sed ''` というコードを実行してみてもらいたい。

■BSD 版 sed の場合

```
$ printf 'Hello,¥nworld!' | sed '' ↵
Hello,
world!
$
```

■GNU 版 sed の場合

```
$ printf 'Hello,¥nworld!' | sed '' ↵
Hello,
world!$
```

と、このように挙動が異なる。最終行が改行コードで終わっていない場合、BSD 版は改行を自動的に挿入し、GNU 版はしないようだ。

純粋なフィルターとして振る舞ってもらいたい場合には GNU 版の方が理想的ではあるが、すべての環境で動くことを目標にするなら BSD 版のような実装の sed とて無視するわけにはいかない。このような sed をはじめ、AWK や grep 等、最終行に改行コードがなければ挿入されてしまうコマンドでの対処法を別のレシピとして記した。→レシピ 1.6（改行無し終端テキストを扱う）参照

使用可能なコマンド・メタ文字

これも、GNU 版は独自拡張されているので注意。

sed の中で使えるコマンドに関して迷ったら、POSIX の sed コマンド[*23]を見る。また、sed で使用可能な正

[*23] http://pubs.opengroup.org/onlinepubs/9699919799/utilities/sed.html

規表現については、正規表現メモさんの記述[*24]が便利だろう。

標準入力指定の “-”

多くのコマンドではファイル名として “-” を指定すると標準入力を意味するのだが、sed ではこれを使ってはならない。BSD 版の sed は、標準入力ではなく真面目に “-” というファイルを開こうとしてエラーになるからだ。

ロケール

→レシピ 5.6（ロケール）を参照

レシピ 5.27　sort コマンド

→レシピ 5.6（ロケール）を参照

レシピ 5.28　tac コマンド・tail コマンド “-r” オプションによる逆順出力

ファイルの行を最後の行から順番に（逆順に）並べたい時は tac コマンドを使うか、tail コマンドの-r オプションのお世話になりたいところであろう。しかし、どちらも一部の環境でしか使えないし、もちろん POSIX にも載っていない。

ではどうするか……。定番は、AWK で行番号を行頭に付けて、数値の降順ソートし、最後に行番号をとるという方法が無難だろう。

■逆順出力するサンプルコード #1

```
#! /bin/sh

# 逆順に並べたいテキストファイル
cat <<TEXT > foo.txt
a
   b
c
TEXT

cat foo.txt          |
awk '{print NR,$0}' | # ←行頭に行番号をつける
sort -k1nr,1         | # ←行番号で降順にソート
sed 's/^[0-9]* //'    # ←行番号を除去
```

また、ソート対象のテキストデータが標準入力ではなくファイルであることがわかっているのであれば、ex コマンドを使うという芸当もある。[*25]

[*24] http://www.kt.rim.or.jp/~kbk/regex/regex.html#SED

[*25] bsdhack 氏のブログ記事 http://blog.bsdhack.org/index.cgi/Computer/20100513.html より引用

■逆順出力するサンプルコード #2

```
#! /bin/sh

# 逆順に並べたいテキストファイル
cat <<TEXT > foo.txt
a
   b
c
TEXT

ex -s foo.txt <<-EOF
  g/^/mo0
  %p
EOF
```

レシピ 5.29　test（“[”）コマンド

どんな内容が与えられるかわからない文字列（シェル変数など）の内容を確認する時、最近の test コマンドなら

■シェル変数$str の内容が “!” ならば “Bikkuri!” を表示

```
[ "$str" = '!' ] && echo 'Bikkuri!'
```

と書いても問題無いものが多い[*26]。しかし、古来の環境では

```
`[: =: unexpected operator`
```

というエラーメッセージが表示され、正しく動作しないものが多い。これは$str に格納されている “!” が、評価すべき文字列ではなく否定のための演算子と解釈され、そうすると後ろに左辺ナシの=が現れたと見なされてエラーになるというわけだ。

test コマンドを用いて、全ての環境で安全に文字列の一致、不一致、大小を評価するには、文字列評価演算子の両辺にある文字列の先頭に無難な一文字を置く必要がある。

■両辺にある文字列の先頭に無難な 1 文字を置けば、どこでも正しく動く

```
[ "_$str" = '_!' ] && echo 'Bikkuri!'
```

もっとも、単に文字列の一致、不一致を評価したいだけなら、test コマンドを使わずに下記のように case 文を使う方がよい。上記のような配慮は必要ないし、外部コマンド（シェルが内部コマンドとして持ってる場合もあるが）の test コマンドを呼び出さなくてよいので軽い。

■case 文で同等のことをする

```
case "$str" in '!') echo 'Bikkuri!';; esac
```

[*26] さすがに$str の中身が “(” だった場合ダメなようだが。

レシピ 5.30　tr コマンド

このコマンドは各環境の方言が強く残るコマンドの一種で、無難に作るならなるべく使用を避けたいコマンドだ。

例えばアルファベットの全ての大文字を小文字に変換したい場合、

```
tr '[A-Z]' '[a-z]' ← System V 系での書式（運よくどこでも動く）
tr 'A-Z' 'a-z'     ← BSD 系、POSIX での書式
```

という 2 つの書式がある。範囲指定の際にブラケット [、] が要るかどうかだ。BSD 系の場合、ブラケットは通常文字として解釈されるので、これを用いると置換対象文字として扱われてしまう。しかしながら前者のブラケットは置換前も置換後も全く同一の文字なので幸いにしてどこでも動く。従って、このようなケースでは前者の記述をとるべきだろう。

しかし、-d オプションで文字を消したい場合はそうはいかない。

```
tr -d '[a-z]' ← System V 系での書式（これは BSD 系、POSIX 準拠実装では NG）
tr 'a-z'      ← BSD 系、POSIX での書式
```

POSIX に準拠してない System V 実装が悪いと言ってしまえばそれまでなのだが、歴史の上では POSIX よりも早いので、それを言うのもまた理不尽というもの。ではどうすればいいか。

答えは、「sed で代用する」だ。上記のように、全ての小文字アルファベットを消したいという場合はこう書けばよい。

```
sed 's/[a-z]//g'
```

レシピ 5.31　trap コマンド

→レシピ 5.21（kill コマンド）参照

第 6 章

レシピを駆使した調理例

Shell Script ライトクックブック第一弾に引き続き、第二弾でも最後にレシピを活用した調理例（サンプルアプリケーション）をご覧に入れよう。今回の料理は、多くのサイトで使われる Web アプリケーション（の部品）である。

本章を読み、シェルスクリプトアプリケーションの速度や実力を見直を見直してもらえれば幸いである。

郵便番号から住所欄を満たすアレをシェルスクリプトで

> 郵便番号を入れ、ボタンを押すと……、都道府県欄から市区町村名欄、町名欄まで満たされ、あとはせいぜい番地を入力すれば住所欄は入力完了。

これはインターネットで買い物をした経験がある方なら殆どの方が体験したことのある機能ではないだろうか。今から作る料理は、この「住所欄補完」アプリケーションである。

アプリケーションの構成

それではまず、構成[*1]から見ていこう。次の表をご覧いただきたい。

[*1] このサンプルアプリケーションは、サンプル品であるという性質上、一切のアクセス制限を掛けていない。実際にアプリケーションを開発する時は、public_html ディレクトリー以外に.htaccess 等のファイルを置いて中を覗かれないようにすべきであろう。

■住所欄補完アプリケーションのファイル構成

```
.
+-- data/  ..................... 郵便番号辞書ファイル関連ディレクトリー
|   |
|   |-- mkzipdic_kenall.sh ... 郵便番号辞書を作成するシェルスクリプト（地域名用）
|   |-- mkzipdic_jigyosyo.sh . 郵便番号辞書を作成するシェルスクリプト（事業所用）
|   |                          ・要 zip コマンド、curl コマンド、及び iconv または nkf コマンド
|   |                          ・crontab などから実行させるとよい
|   |
|   +-- kenall.txt ........... 辞書ファイル（地域名用、mkzipdic_kenall.sh によって生成される）
|   +-- jigyosyo.txt ......... 辞書ファイル（事業所用、mkzipdic_jigyosyo.sh によって生成される）
|
|
+-- public_html/ ............. Web ディレクトリー（httpd でこの中を公開する）
|   |
|   +-- index.html  .......... 入力フォーム（Web ブラウザーでこのファイルを開く）
|   +-- zip2addr.js .......... 郵便番号→住所 変換用クライアントサイドプログラム
|   +-- zip2addr.ajax.cgi .... 郵便番号→住所 変換用サーバーサイドプログラム
|
|
+-- commands ................. 自作コマンド置き場
    |
    +-- parsrc.sh ............ CSV パーサー
```

自作コマンドである CSV パーサー[*2]を置いてあるディレクトリー “commands” 以外に、2 つのディレクトリー（“data” と “public_html”）がある。これは、住所欄補完という機能を実現するにはやるべき作業が 2 種類あることに理由がある。では、それぞれについて説明しよう。

data ディレクトリー – 住所辞書作成

1 つ目の作業は、辞書づくりである。

郵便番号に対応する住所の情報は、日本郵政のサイトで公開されているが、クライアント（Web ブラウザー）から郵便番号を与えられる度にそれを見にいくのは効率が悪い。そこで、その情報を手元にダウンロードしておくのだ。

しかし単にダウンロードするだけではない。圧縮ファイルになっているので回答するのはもちろんだが、Shift_JIS エンコードされた CSV ファイルとしてやってくるうえに、よみがな等の今回の変換に必要の無いデータもあるためそのままの状態では扱いづらい。そこで、UTF-8 へエンコードし、CSV ファイルをパースし、郵便番号と住所（都道府県名、市区町村名、町名）という情報だけにした状態で辞書ファイルにしておく。こうすることで、毎回の住所検索が低負荷で高速にこなせるようになる。

この作業を担うのが、data ディレクトリーの中にある “mkzipdic_kenall.sh”、“mkzipdic_jigyosyo.sh” という 2 つのシェルスクリプトだ。2 つあるのは、日本郵政サイトにある辞書データが、一般地域名用と大口事業所名用という 2 つのファイルに分かれているからである。

public_html ディレクトリー – 住所補完処理

前述の作業で作成された辞書ファイルを用い、クライアントから与えられた郵便番号に基づいた住所を住所欄に埋めるのがこのディレクトリーの中にあるプログラムの作業である。

“index.html” は住所欄を提供する HTML で、“zip2addr.js” は入力された郵便番号のサーバーへの送信・結

[*2] レシピ 3.5（CSV ファイルを読み込む）参照

果の住所欄への入力を担当する JavaScript だ。そして、受け取った 7 桁の郵便番号から辞書を引き、得られた住所文字列を返すシェルスクリプトが “zip2addr.ajax.cgi” である。

名前を見ればわかるがこのシェルスクリプトも Ajax として動作するので、レシピ 4.6（Ajax で画面更新したい）に従って部分 HTML を返してもよいのだが、ここでは敢えて JSON 形式で返すことにした。「もちろん JSON で返すこともできる」ということを示すためだ。JSON で返せば、例えばクライアント側で何らかの汎用 JavaScript ライブラリーを利用していて、それと繋ぎ込むといったことも可能というわけだ。

ソースコード

概要が掴めたところで、主要なソースコードを記していくことにする。シェルスクリプトで構成された Web アプリケーションの中身を、とくと堪能してもらいたい。

尚、これらのソースコードは GitHub でも公開している[*3]。

■ data/mkzipdic_kenall.sh – 辞書ファイル作成（一般地域名用）

このプログラムは、Web サイトから ZIP ファイルをダウンロードして展開する都合により、POSIX 非準拠の curl コマンドと unzip コマンドを必要とすることを御了承願いたい。

```
#! /bin/sh

#########################################################################
#
# MKZIPDIC_KENALL.SH
# 日本郵便公式の郵便番号住所 CSV から、本システム用の辞書を作成（地域名）
#
# Usage : mkzipdic.sh -f
#         -f ... ・サイトにある CSV ファイルのタイプスタンプが、
#                  今ある辞書ファイルより新しくても更新する
#
# [出力]
# ・戻り値
#   - 作成成功もしくはサイトのタイムスタンプが古いために作成する必要無
#     しの場合は 0、失敗したら 0 以外
# ・成功時には辞書ファイルを更新する。
#
#########################################################################


#########################################################################
# 初期設定
#########################################################################

# --- 変数定義 -----------------------------------------------------
dir_MINE="$(d=${0%/*}/; [ "_$d" = "_$0/" ] && d='./'; cd "$d"; pwd)" # この sh のパス
readonly file_ZIPDIC="$dir_MINE/ken_all.txt"                        # 郵便番号辞書ファイルのパス
readonly url_ZIPCSVZIP=http://www.post.japanpost.jp/zipcode/dl/oogaki/zip/ken_all.zip
                                                                    # 日本郵便 郵便番号-住所
                                                                    # CSV データ（Zip 形式）URL
readonly flg_SUEXECMODE=0                                           # サーバーが suEXEC モードで
                                                                    # 動いているなら 1 を設定
```

[*3] https://github.com/ShellShoccar-jpn/zip2addr

```
# --- ファイルパス ----------------------------------------------------
PATH='/usr/local/tukubai/bin:/usr/local/bin:/usr/bin:/bin'

# --- 終了関数定義（終了前に一時ファイル削除） ------------------------
exit_trap() {
  trap 0 1 2 3 13 14 15
  [ -n "${tmpf_zipcsvzip:-}" ] && rm -f $tmpf_zipcsvzip
  [ -n "${tmpf_zipdic:-}"    ] && rm -f $tmpf_zipdic
  exit ${1:-0}
}
trap 'exit_trap' 0 1 2 3 13 14 15

# --- エラー終了関数定義 ----------------------------------------------
error_exit() {
  [ -n "$2" ] && echo "${0##*/}: $2" 1>&2
  exit_trap $1
}

# --- テンポラリーファイル確保 ----------------------------------------
tmpf_zipcsvzip=$(mktemp -t "${0##*/}.XXXXXXXX")
[ $? -eq 0 ] || error_exit 1 'Failed to make temporary file #1'
tmpf_zipdic=$(mktemp -t "${0##*/}.XXXXXXXX")
[ $? -eq 0 ] || error_exit 2 'Failed to make temporary file #2'



########################################################################
# メイン
########################################################################

# --- 引数チェック ----------------------------------------------------
flg_FORCE=0
[ ¥( $# -gt 0 ¥) -a ¥( "_$1" = '_-f' ¥) ] && flg_FORCE=1

# --- cURL コマンド存在チェック ---------------------------------------
which curl >/dev/null
[ $? -eq 0 ] || error_exit 3 'curl command not found'

# --- サイト上のファイルのタイムスタンプを取得 ------------------------
timestamp_web=$(curl -sLI $url_ZIPCSVZIP                                |
                awk '                                                    #
                  BEGIN{                                                 #
                    status = 0;                                          #
                    d["Jan"]="01";d["Feb"]="02";d["Mar"]="03";d["Apr"]="04"; #
                    d["May"]="05";d["Jun"]="06";d["Jul"]="07";d["Aug"]="08"; #
                    d["Sep"]="09";d["Oct"]="10";d["Nov"]="11";d["Dec"]="12"; #
                  }                                                      #
                  /^HTTP¥// { status = $2; }                             #
                  /^Last-Modified/ {                                     #
                    gsub(/:/, "", $6);                                   #
                    ts = sprintf("%04d%02d%02d%06d" ,$5,d[$4],$3,$6);    #
                  }                                                      #
                  END {                                                  #
                    if ((status>=200) && (status<300) && (length(ts)==14)) { #
                      print ts;                                          #
                    } else {                                             #
```

```
                    print "NOT_FOUND";                                    #
                  }                                                       #
                }'                                                        )
[ "$timestamp_web" != 'NOT_FOUND' ] || error_exit 4 'The zipcode CSV file not found on the web'
echo "_$timestamp_web" | sed '1s/_//' | grep '^[0-9]¥{14¥}$' >/dev/null
[ $? -eq 0 ] || timestamp_web=$(TZ=UTC/0 date +%Y%m%d%H%M%S) # 取得できなければ現在日時を入れる

# --- 手元の辞書ファイルのタイムスタンプと比較し、更新必要性確認 -----
while [ $flg_FORCE -eq 0 ]; do
  # 手元に辞書ファイルはあるか?
  [ ! -f "$file_ZIPDIC" ] && break
  # その辞書ファイル内にタイムスタンプは記載されているか?
  timestamp_local=$(head -n 1 "$file_ZIPDIC" | awk '{print $NF}')
  echo "_$timestamp_local" | sed '1s/_//' | grep '^[0-9]¥{14¥}$' >/dev/null
  [ $? -eq 0 ] || break
  # サイト上のファイルは手元のファイルよりも新しいか?
  [ $timestamp_web -gt $timestamp_local ] && break
  # そうでなければ何もせず終了(正常)
  exit 0
done

# --- 郵便番号 CSV データファイル(Zip 形式)ダウンロード -----------------
curl -s $url_ZIPCSVZIP > $tmpf_zipcsvzip
[ $? -eq 0 ] || error_exit 5 'Failed to download the zipcode CSV file'

# --- 郵便番号辞書ファイル作成 ----------------------------------------
unzip -p $tmpf_zipcsvzip                                              |
# 日本郵便 郵便番号-住所 CSV データ(Shift_JIS)                       #
if   which iconv >/dev/null; then                                       #
  iconv -c -f SHIFT_JIS -t UTF-8                                        #
elif which nkf   >/dev/null; then                                       #
  nkf -Sw80                                                             #
else                                                                    #
  error_exit 6 'No KANJI convertors found (iconv or nkf)'               #
fi                                                                      |
# 日本郵便 郵便番号-住所 CSV データ(UTF-8 変換済)                    #
$dir_MINE/../commands/parsrc.sh                                         | # CSV パーサー(自作コマンド)
# 1:行番号 2:列番号 3:CSV データセルデータ                           #
awk '$2~/^3|7|8|9$/'                                                    |
# 1:行番号 2:列番号(3=郵便番号,7=都道府県,8=市区町村,9=町) 3:データ
awk 'BEGIN{z="#"; p="generated"; c="at"; t="'$timestamp_web'"; }  #
     $1!=line      {pl();z="";p="";c="";t="";line=$1;          }  #
     $2==3         {z=$3;                                      }  #
     $2==7         {p=$3;                                      }  #
     $2==8         {c=$3;                                      }  #
     $2==9         {t=$3;                                      }  #
     END           {pl();                                      }  # #   地域名住所文字列で
     function pl() {print z,p,c,t;                             }' | #   小括弧以降は
sed 's/(.*//'                                                           | # ←使えないので除去する
sed 's/以下に.*//'                                                      > $tmpf_zipdic #"以下に"も同様
# 1:郵便番号 2:都道府県名 3:市区町村名 4:町名
[ -s $tmpf_zipdic ] || error_exit 7 'Failed to make the zipcode dictionary file'
mv $tmpf_zipdic "$file_ZIPDIC"
[ "$flg_SUEXECMODE" -eq 0 ] && chmod go+r "$file_ZIPDIC" # suEXEC で動いていない場合は
                                                         # httpd にも読めるようにする
```

```
##########################################################################
# 正常終了
##########################################################################

exit 0
```

■ public_html/index.html – 入力フォーム

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="ja">

<haed>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<meta http-equiv="Content-Style-Type" content="text/css" />
<style type="text/css">
<!--
    dd { margin-bottom: 0.5em; }
    #addressform { width: 50em; margin: 1em 0; padding: 1em; border: 1px solid; }
    #inqZipcode1,#inqZipcode2 {font-size: large; font-weight: bold;}
    .type_desc {font-size: small; font-weight: bold;}
-->
</style>
<meta http-equiv="Content-Script-Type" content="text/javascript" />
<script type="text/javascript" src="zip2addr.js"></script>
<title>郵便番号→住所検索 Ajax by シェルスクリプト デモ</title>
</haed>


<body>
<h1>郵便番号→住所検索 Ajax by シェルスクリプト デモ</h1>

<form action="#dummy">

<table border="0"  id="addressform">
  <tr>
    <td colspan="3">
      <dl>
        <dt>郵便番号</dt>
        <dd><input id="inqZipcode1" type="text" name="inqZipcode1" size="3" maxlength="3" />
            -
            <input id="inqZipcode2" type="text" name="inqZipcode2" size="4" maxlength="4" />
        </dd>
      </dl>
    </td>
  </tr>

  <tr>
    <td>
      <dl>
        <dt>住所検索<br /></dt>
        <dd><input id="run" type="button" name="run" value="実行" onclick="zip2addr();" /></dd>
        <dt>住所 (都道府県名)</dt><dd>
                                    <select id="inqPref" name="inqPref">
                                      <option>(選択してください)</option>
```

```
                                    <option>北海道</option>
                                              :
                                    <option>沖縄県</option>
                                  </select>
                                </dd>
        <dt>住所 (市区町村名)</dt>
          <dd><input id="inqCity" type="text" size="20" name="inqCity" /></dd>
        <dt>住所 (町名)</dt>
          <dd><input id="inqTown" type="text" size="20" name="inqTown" /></dd>
      </dl>
    </td>
  </tr>
</table>

</form>

</body>

</html>
```

■ public_html/zip2addr.js – 住所補完（クライアント側）

```
// ===== Ajax のお約束オブジェクト作成 ================================
// [入力]
// ・なし
// [出力]
// ・成功時: XmlHttpRequest オブジェクト
// ・失敗時: false
function createXMLHttpRequest(){
  if(window.XMLHttpRequest){return new XMLHttpRequest()}
  if(window.ActiveXObject){
    try{return new ActiveXObject("Msxml2.XMLHTTP.6.0")}catch(e){}
    try{return new ActiveXObject("Msxml2.XMLHTTP.3.0")}catch(e){}
    try{return new ActiveXObject("Microsoft.XMLHTTP")}catch(e){}
  }
  return false;
}


// ===== 郵便番号による住所検索ボタン ================================
// [入力]
// ・HTML フォームの、id="inqZipcode1"と id="inqZipcode2"の値
// [出力]
// ・指定された郵便番号に対応する住所が見つかった場合
//   - id="inqPref"な<select>の都道府県を選択
//   - id="inqCity"な<input>に市区町村名を出力
//   - id="inqTown"な<input>に町名を出力
// ・見つからなかった場合は alert メッセージ
function zip2addr() {
  var sUrl_to_get;  // 汎用変数
  var sZipcode;     // フォームから取得した郵便番号文字列の格納用
  var xhr;          // XML HTTP Request オブジェクト格納用
  var sUrl_ajax;    // Ajax の URL 定義用

  // --- 1) 呼び出す Ajax CGI の設定 ----------------------------------
```

```
  sUrl_ajax = 'zip2addr.ajax.cgi';

  // --- 2) 郵便番号を取得する ----------------------------------------
  if (! document.getElementById('inqZipcode1').value.match(/^([0-9]{3})$/)) {
    alert(' 郵便番号 (前の 3 桁) が正しくありません');
    return;
  }
  sZipcode = "" + RegExp.$1;
  if (! document.getElementById('inqZipcode2').value.match(/^([0-9]{4})$/)) {
    alert(' 郵便番号 (後の 4 桁) が正しくありません');
    return;
  }
  sZipcode = "" + sZipcode + RegExp.$1;


  // --- 3)Ajax コール ------------------------------------------------
  xhr = createXMLHttpRequest();
  if (xhr) {
    sUrl_to_get  = sUrl_ajax;
    sUrl_to_get += '?zipcode='+sZipcode;
    sUrl_to_get += '&dummy='+parseInt((new Date)/1); // ブラウザ cache 対策
    xhr.open('GET', sUrl_to_get, true);
    xhr.onreadystatechange = function(){zip2addr_callback(xhr, sAjax_type)};
    xhr.send(null);
  }
}
function zip2addr_callback(xhr, sAjax_type) {

  var oAddress;     // サーバーから受け取る住所オブジェクト
  var e;            // 汎用変数 (エレメント用)
  var sElm_postfix; // 住所入力フォームエレメント名の接尾辞格納用

  // --- 4) 住所入力フォームエレメント名の接尾辞を決める --------------
  switch (sAjax_type) {
    case 'API_XML'  : sElm_postfix = '_API_XML' ; break;
    case 'API_JSON' : sElm_postfix = '_API_JSON'; break;
    default         : sElm_postfix = ''         ; break;
  }

  // --- 5) アクセス成功で呼び出されたのでないなら即終了 --------------
  if (xhr.readyState != 4) {return;}
  if (xhr.status == 0    ) {return;}
  if      (xhr.status == 400) {
    alert(' 郵便番号が正しくありません');
    return;
  }
  else if (xhr.status != 200) {
    alert(' アクセスエラー (' + xhr.status + ')');
    return;
  }

  // --- 6) サーバーから返された住所データを格納 ----------------------
  oAddress =  JSON.parse(xhr.responseText);
  if (oAddress['zip'] === '') {
    alert(' 対応する住所が見つかりませんでした');
    return;
```

```
  }

  // --- 7) 都道府県名を選択する ------------------------------------
  e = document.getElementById('inqPref'+sElm_postfix)
  for (var i=0; i<e.options.length; i++) {
    if (e.options.item(i).value == oAddress['pref']) {
      e.selectedIndex = i;
      break;
    }
  }

  // --- 8) 市区町村名を流し込む ------------------------------------
  document.getElementById('inqCity'+sElm_postfix).value = oAddress['city'];

  // --- 9) 町名を流し込む ------------------------------------------
  document.getElementById('inqTown'+sElm_postfix).value = oAddress['town'];

  // --- 99) 正常終了 -----------------------------------------------
  return;
}
```

■ public_html/zip2addr.ajax.cgi – 住所補完（サーバー側）

```
#! /bin/sh

########################################################################
#
# ZIP2ADDR.AJAX.CGI
# 郵便番号ー住所検索
#
# [入力]
# ・[CGI 変数]
#   - zipcode: 7 桁の郵便番号（ハイフン無し）
# [出力]
# ・成功すれば JSON 形式で郵便番号、都道府県名、市区町村名、町名
# ・郵便番号辞書ファイル無し→ 500 エラー
# ・郵便番号指定が不正        → 400 エラー
# ・郵便番号が見つからない   →空文字の JSON を返す
#
########################################################################


########################################################################
# 初期設定
########################################################################

# --- 変数定義 -----------------------------------------------------
dir_MINE="$(d=${0%/*}/; [ "_$d" = "_$0/" ] && d='./'; cd "$d"; pwd)" # この sh のパス
readonly file_ZIPDIC_KENALL="$dir_MINE/../data/ken_all.txt"          # 辞書（地域名）のパス
readonly file_ZIPDIC_JIGYOSYO="$dir_MINE/../data/jigyosyo.txt"       # 辞書（事業所名）パス

# --- ファイルパス -------------------------------------------------
PATH='/usr/local/bin:/usr/bin:/bin'

# --- エラー終了関数定義 -------------------------------------------
```

```
error500_exit() {
  cat <<-__HTTP_HEADER
  Status: 500 Internal Server Error
  Content-Type: text/plain
  500 Internal Server Error
  ($@)
__HTTP_HEADER
  exit 1
}
error400_exit() {
  cat <<-__HTTP_HEADER
  Status: 400 Bad Request
  Content-Type: text/plain
  400 Bad Request
  ($@)
__HTTP_HEADER
  exit 1
}


########################################################################
# メイン
########################################################################

# --- 郵便番号データファイルはあるか？ --------------------------------
[ -f "$file_ZIPDIC_KENALL"   ] || error500_exit 'zipcode dictionary #1 file not found'
[ -f "$file_ZIPDIC_JIGYOSYO" ] || error500_exit 'zipcode dictionary #2 file not found'

# --- CGI 変数 (GET メソッド) で指定された郵便番号を取得 ------------------
zipcode=$(echo "_${QUERY_STRING:-}" | # 環境変数で渡ってきた CGI 変数文字列を STDOUT へ
          sed '1s/^_//'             | # echo の誤動作防止のために付けた"_"を除去
          tr '&' '¥n'               | # CGI 変数文字列 (a=1&b=2&...) の&を改行に置換し、1 行 1 変数に
          grep '^zipcode='          | # 'zipcode' という名前の CGI 変数の行だけ取り出す
          sed 's/^[^=]¥{1,¥}=//'    | # "CGI 変数名="の部分を取り除き、値だけにする
          grep '^[0-9]¥{7¥}$'       ) # 郵便番号の書式の正当性確認

# --- 郵便番号はうまく取得できたか？ ----------------------------------
[ -n "$zipcode" ] || error400_exit 'invalid zipcode'

# --- JSON 形式文字列を生成して返す ------------------------------------
cat "$file_ZIPDIC_KENALL" "$file_ZIPDIC_JIGYOSYO"                  | # 辞書ファイルを開く
#  1:郵便番号 2〜:各種住所データ                                  #
awk '$1=="'$zipcode'"{hit=1;print;exit} END{if(hit==0){print ""}}' | # 該当行を取出し (1 行のみ)
while read zip pref city town; do # HTTP ヘッダーと共に、JSON 文字列化した住所データを出力する
  cat <<-__HTTP_RESPONSE
  Content-Type: application/json; charset=utf-8
  Cache-Control: private, no-store, no-cache, must-revalidate
  Pragma: no-cache
  {"zip":"$zip","pref":"$pref","city":"$city","town":"$town"}
__HTTP_RESPONSE
  break
done

# --- 正常終了 --------------------------------------------------------
exit 0
```

動作画面

実際の動作画面を掲載する。尚、デモページ[*4]も用意している。

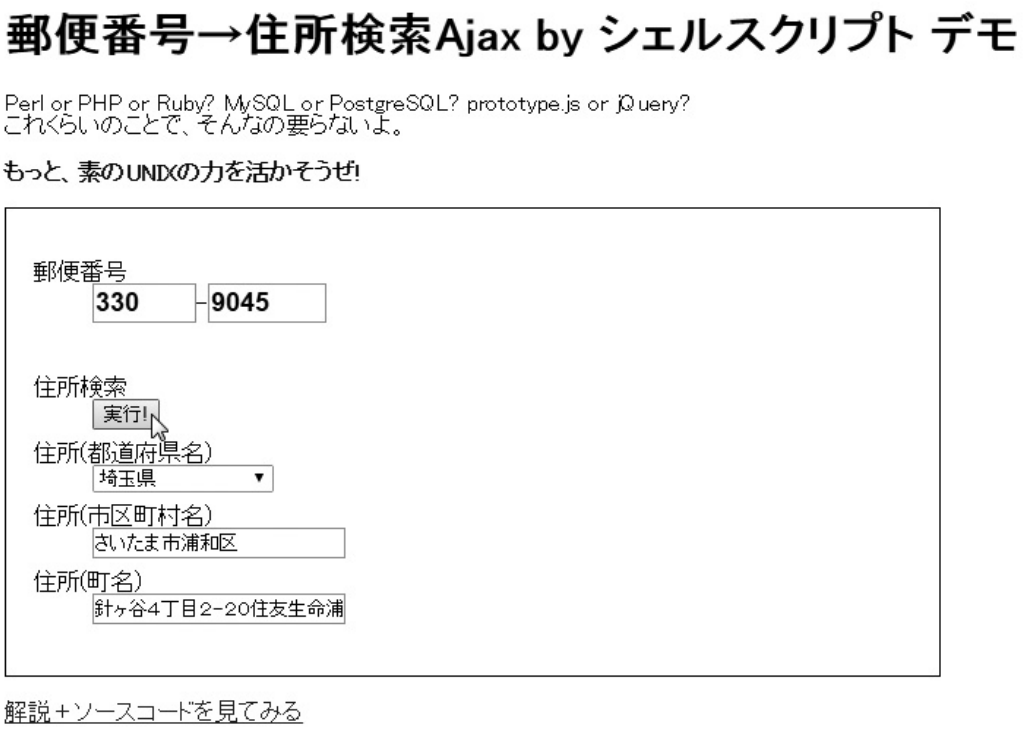

図 6.1 住所補完アプリケーションの動作画面

使い心地（速度）はいかがだろうか。ちなみに**辞書データは地域と事業所を併せ、およそ 14 万件**である。シェルスクリプトで開発したアプリケーションであっても、これだけの速度で動くということを実感し、「シェルスクリプトなんてプログラム開発には使えない」という思い込みは捨て去ってもらえれば大変うれしい。

[*4] http://lab-sakura.richlab.org/ZIP2ADDR/public_html/

あとがき

● カバーの説明

本書の表紙の動物はユリカモメです。カモメの一種ですが、くちばしと脚が赤いのが特徴です。古くは都鳥（みやこどり）とも呼ばれたようです。

渡り鳥であり、日本には 11 月頃にやってきて 4 月頃まで越冬のため滞在します。このイラストは、東京の川の止まり木に一人（一鳥）佇む冬羽のユリカモメです。

そして頭部の黒い夏羽に生え変わると、より高緯度の地域へ、繁殖のために旅立ちます。北米などでは、夏羽になった夏季に見られるため、ユリカモメの英語名は “black-headed gull”（直訳すればクロアタマカモメ）といいます。

ところで、「ユリカモメ」と聞いて乗り物しか思い浮かばない人は、鉄分過多、あるいは有明病の疑いがありますのでご注意を。

● 著者コメント

リッチ・ミカン
同人活動を始めたのが 2002 年で、気が付けばもう 12 年も続いている。10 年目には名著 “sed & AWK” の著者でオ○イリー創業者の一人にインタビュー（ついでに小誌を見せる）という奇跡まで果たしたが、始めた頃にこんな未来が想像できただろうか？いいや、一発屋でせいぜい数年の同人作家生活だと思っていた。

時代の移り変わりが特に激しいコンピューターの分野で同人作家が続いているからには、12 年前には聞きもしなかった新技術を話題にした本を書いているかと思いきや！……今書いているのは 1970 年代に誕生し、1990 年頃に POSIX として標準型がまとめられた UNIX である。しかもその「POSIX 原理主義を受け入れよ」と言っている。一番最初に書いた本は 1983 年生まれの MSX パソコンの話であったから、時代が進むどころか遡っている。同人活動を始めた頃にこんな未来が想像できただろうか？いいや、新技術についていけずに 35 歳を待たずしてコンピューターエンジニアを引退するものと思っていた。

今から 12 年後は何をやっているんだろうか？この調子ならひょっとすると 10 年後にティム・オ○イリーに会って、計算尺やそろばん、あるいはクルタ計算機の同人誌を見せびらかしているかもしれない。

E-mail : richmikan@richlab.org

● 表紙担当者コメント

もじゃ
久しぶりの登場もじゃである。

普段はもっぱらもじゃもじゃしている、もじゃ SE である。実は密かに思い悩み、メンズ脱毛のカウンセリングに行ったところ、「成功する保証はない」と言われ、「ならばせめて成功率を教えてくれ」と語気を荒げて迫ったが、「わからない」とすげなく言われてしまい、ブチギレて帰ってきた次第である。理系としては成功率の統計もとっていないようなところは信用できないのである。

それはそうと、誰にも同意してもらえないが、最近は人差し指と薬指の区別がつかなくて困っている。

Shell Script ライトクックブック 2014 — POSIX 原理主義を貫く

2014 年 12 月 30 日　初版発行

著　　者	リッチ・ミカン
表　　紙	もじゃ
制 作 協 力	321516（三井浩一郎）
印刷・製本	株式会社イニュニック
発行・発売	松浦リッチ研究所
	http://richlab.org/
影の発行元	秘密結社シェルショッカー日本支部

Printed in Japan