

Shell Script ライトクックブック

2014 改訂版

リッチ・ミカン 著



まえがき

デプロイやその後の保守に苦しむプログラマーへ

これはシェルスクリプト (Bourne Shell) のレシピ集である。だが単なるレシピ集なら既にいくらかでも存在する。本書は、UNIX 系 OS 向けアプリケーションを開発し、それをデプロイ (インストール)、或いはその後保守する人々が苦しまないようにするためのプログラミングに役立つレシピを主に取り揃えた。このような本を作るに至った経緯について、少し説明させてもらいたい。

とある会社との出会い

私は元々 UNIX やシェルスクリプトが好きで、それをネタに同人誌を作っていた。

2010 年、その同人誌をイベント会場で出展していたとある会社の担当者に見せた。どんな仕事をやっているのかと尋ねたら、いわゆるデータベース (RDB) を使わず、シェルスクリプトだけでシステムを作り、しかもそれで商売が成り立っているという話だった。それは UNIX ユーザーにとってとても理想的な姿ではあるが、そんなことが本当にできるのか疑問だった。

しかし、説明を聞けば聞く程信じるようになった。UNIX 哲学に根ざした設計思想やシステム開発上の指針、そして結果として作られたシステムがその理屈通り、高いパフォーマンスを実際に叩きだしており、なんとも痛快だった。彼らは自分達が確立したこれら独自の開発手法に名前まで付けていた。

痛い目に散々遭ってきた人たち

彼らの言ってる事が嘘ではないと思えたもう一つの理由は、披露された苦労話にとっても現実味を感じられたからだった。やはり当初、このやり方が使えるということを説得するのが大変だったらしい。また、その開発手法が切れの味良い開発手法である秘密の一つに洗練された独自コマンド群を提供しているという特徴があるが、当初は 2000 個以上作ったものの使いきれずに淘汰が進み、本当に洗練された数十個だけが残ったという。確かに想像に難しくない話だ。

そんな苦労話の中で私が特に感銘を受けたのが、数々のハードウェアや UNIX 系 OS を渡り歩かざるを得なかった時のエピソードだった。顧客の要望で、使えるハードウェアや OS が指定されたり、変更されたりすることは珍しくなかったという。また入手性の問題により、異なるハードウェアや OS のマシンでシステムを構成せざるを得ないということもあったそうだ。こんな状況に置かれた彼らがどんな痛い目に遭うか。これまた想像に難しくない。

どこでも使えるコードしか書かない

使えるコマンドやオプションの違いはもちろん、扱うデータの数値フォーマットの違いや文字コードの違い (Shift-JIS や EUC-JP、ASCII、EBCDIC 等々) など、開発したソフトを別種の OS で動かすたびに、いちいち躓いて痛い目に遭ってきたという。データは単に変換すれば解決するわけではない。例えば文字データであれば、変換先でどうしてもうまく扱えない文字というのが稀にあり、実際に動かし始めてその存在を知ることになったという。

こういった数々の問題に悩まされた結果として彼らが身に付けた教訓は「どこでも使えるコードしか書かない」、「問題を引き起こしそうなデータは避ける (別の方法で表現する)」だった。一見単純な発想に思えるが、私はそこに大きな感銘を受けた。普通の人なら「互換性がないからこの言語は使えない」といって使用を諦めてしまふところだが、彼らはその単純な発想を実践し、実際に今日まで乗り切ってきたという事実に対してだ。

POSIX 原理主義のススメ

この単純明快な発想に感銘を受けた私は、気が付けば彼らよりもエキセントリックにその発想を実践するようになっていた。それが POSIX 原理主義という考え方だ。

POSIX とは

Portable Operating System Interface の略であるとされる。X が無いが、本来末尾にあった “for uniX” が取れたのだろう (と、勝手に解釈している)。POSIX とは、UNIX 系 OS 同士が互換性を持つために各々が守るべき表面部分の仕様をまとめた規格である。1990 年、既に様々な系譜が存在していた UNIX 系 OS において、それでも共通している仕様をできるだけ抜き出しながら、「これさえ守れば UNIX を名乗る OS で互換性のあるものが作れる (だから皆で守ろう)」という最低限の規格として IEEE によってまとめられたものだと、私は理解している。その後も内容が更新されて現在に至っているものの、「最低限これだけは皆で守ろう」という立場ゆえに安易に変えるわけにはいかず、非常にゆっくりとした改訂がなされている。

OS 乗り換えもバージョンアップも怖くない

POSIX に書かれている内容を大雑把に言うなら man である。つまり、コマンドやシステムコールの仕様、データフォーマット等、ユーザーが意識すべき仕様がまとめられている。実在する各 OS の man はこれを元にして作られているといっても過言ではないだろう。従って POSIX に明記されている仕様の範囲でソフトウェアを作れば、UNIX を名乗る OS になら殆ど何も恐れることなく持ち運ぶことができる。

また、OS のバージョンアップも特に恐れなくていい。OS 開発元からバージョンアップ勧告が出たら恐れずやればいいし、強制的にバージョンアップがなされるレンタルサーバーにも安心して置ける。なぜなら「最低限これだけは皆で守ろう」という仕様からなる規格なのだから、OS 開発元も POSIX に書かれている仕様だけは、例え他の仕様を変更しても極力維持しようとする。UNIX 系 OS であり続けるために。逆に、セキュリティ脆弱性が見つかったのにいつまでたっても OS 開発元が修正版をリリースしないというなら躊躇なく別 OS に引越すこともできる。なぜなら引越し先の OS も POSIX に書かれている仕様は極力守っているからだ。もちろん POSIX の仕様が変わってしまったら元も子も無いのだが、先程も述べたように POSIX は「最低限これだけは皆で守ろう」という仕様からなる規格であるために易々と変更を促すわけにもいかず、結果として 10 年、20 年の規模で通用する規格になっている。

OS 乗り換えにもバージョンアップにも強いということは、つまり場所も時代も問わないということである。このような性質を私は「時空を超えた可搬性」と呼んでいるが、POSIX にはそれが秘められているのである。

POSIX 原理主義の言語 – Bourne シェルスクリプト

本書はシェルスクリプトのレシピ集であると冒頭で述べたが、数ある言語の中でなぜシェルスクリプト (bash その他ではなく Bourne Shell) なのかここでようやく説明することができる。理由は POSIX 規格に存在する数少ない言語の一つだからである。POSIX に存在するということは、先に述べた「時空を超えた可搬性」を享受できるということだ。もちろん POSIX の範囲のシェル文法とコマンドだけを使うことが前提ではあるが。

巷ではよく「シェルスクリプトは環境依存が激しいから……」などと敬遠され、書き捨てるプログラムのための言語とみなされる。だが、書き捨てず、10 年、20 年持たせるに相応しい言語こそシェルスクリプトだ。

因みに POSIX 規格で存在する言語には他に C 言語 (C99) もある。よって C 言語を使っても POSIX 原理主義は実践できるのだが、例えばポインタの概念を理解しなければならなかったりコンパイルが必要だったり、手軽さではシェルスクリプトに及ばないのが難点だ。開発者にとっては危険なバグが入り込みやすいし、運用者にとってみればインストール対象のコンピューター上でコンパイルが通らずに何時間もの格闘を余儀なくされることなど珍しくない。これは C という言語が、時にコンピューターの内部構造を深く意識しながら書かねばならない言語であることに起因する。

もちろん欠点ばかりではない。C 言語は、シェルスクリプトには太刀打ちできない処理速度を持っている。ただ、シェルスクリプトにおいても sed や AWK 他、既存の POSIX コマンドを上手に活用すれば実用的な速度は十分発揮できる。なぜなら既存のコマンドは、どれも C 言語で作られているからだ。この後で紹介する実際の制作アプリケーションを試してもらえば、シェルスクリプトで作っても申し分無い処理速度が出せることに納得してもらえと思う。

以上、シェルスクリプトによる **POSIX 原理主義**の利点をまとめると次のとおりだ。

- どの UNIX 系 OS でも動く
- 10 年、20 年先も動く
- コピー一発、デプロイ完了

POSIX 原理主義に基づくアプリケーションたち

POSIX 原理主義は机上の空論などではない。既にこの主義を実践して制作したアプリケーションがいくつかあるが、代表的な 2 つを紹介しよう。どちらも、POSIX 原理主義に基づくシェルスクリプトで書かれている。

ショッピングカート「シェルショッカー 1 号男」

私はもともと同人作家であるので年二回の例のイベント (コミックマーケット) で本を頒布している。そして会場に来られない人向けに通販もやっている。ショッピングカートプログラムが必要になるところだが、言語は Perl や PHP で、データベースとして MySQL を必要とするといったいわゆる LAMP 環境のものしかない。一つの同人サークルとして細々やりたいたいだけにデータベースは大げさだし、同人誌頒布に特化した機能がなくて使いづらいし、そもそもシェルスクリプトでシステム開発する本を頒布するのに Perl や PHP その他を使うなんて何のジョークだ! ということで作ったものがシェルショッカー 1 号男 (=シェルスクリプト製ショッピングカート version1 の意味) だ。

本書を買ってくれた方にはもはや不要だが、冷やかに是非

<http://richlab.org/coterie/ssr2.html>

を訪れてみてもらいたい。(注文する前までの操作ならタダ) 注文までいかないとお目に掛かれないが、PayPal API と連携してクレジットカード決済することも可能だ。ソースコードも GitHub で公開している*1。



図1 シェルショッカー 1 号男 (ショッピングカート) を搭載したページ

■POSIX 原理主義と実利のトレードオフ ツッコミが来ると思うので予め断っておくが、郵便番号を住所に変換するデータを取得するために内部では POSIX にはない curl コマンドや、注文確認メールを送るためにこれまた POSIX 外の sendmail コマンドを使っており、そしてそもそも Web (CGI) を使っている時点 POSIX を逸脱しているだろうと指摘されても仕方がない。実は POSIX には、通信系のコマンドが殆ど用意されていないという弱点がある。POSIX 規格に完璧に遵守しようと思ったら、現状では C 言語で curl や sendmail や httpd サーバーを再発明しない限り不可能である。

それらに関しては実利を取ることを甘受するが、それでも代替品があって容易に乗り換えられるという担保は確保している。curl の代替品としては wget や FreeBSD の fetch コマンドを検討し、sendmail コマンドはオリジナルの sendmail サーバーの他、Postfix や qmail、exim など、主要 MTA には同等のコマンドが存在するし、Web サーバーにも Apache の他、nginx や lighttpd 等代替りのものにすぐに移れるように作り、またバージョンアップが起こっても極力影響を受けないように基本的な機能しか使わないようにして作るのである。

*1 <https://github.com/ShellShoccar-jpn/shellshoccar1>

東京メトロ列車在線状況確認アプリ「メトロパイパー」

2014 年、東京メトロが駅施設情報や列車情報を JSON 形式で出力する Web API をオープンデータとして公開し、これを活用するコンテストを開催した。本アプリケーションは、そのコンテストにシェルスクリプトで挑んだ応募作品である。

<http://metropiper.com/>

メトロパイパーは、東京メトロの車両全車両が現在どの駅（駅間）に在線しているかという情報（在線情報）を問い合わせ、JSON で返された在線情報を読み解き、HTML 化して画面に表示するというものである。POSIX の範囲には通信系コマンドが殆ど無いという事情により、Web API へのアクセスには curl コマンドを使っているものの、JSON の解析は sed や AWK を駆使して独自に行っている（→レシピ 3.6 参照）。また、こちらもソースコードを GitHub で公開している*2。

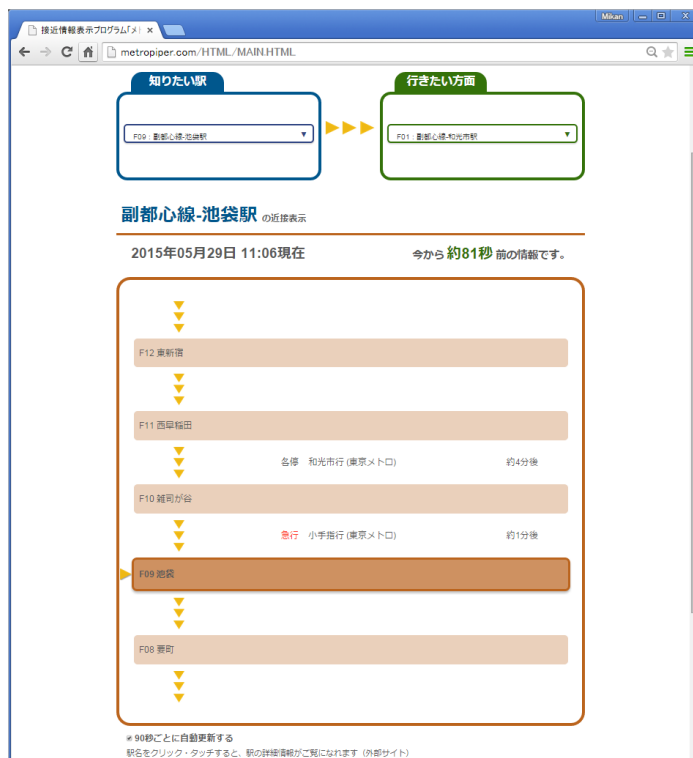


図 2 メトロパイパーの動作画面

*2 <https://github.com/ShellShoccar-jpn/metropiper>

■何十万もの人々が路頭に迷わないために メトロパイパーの最大の特長は、10年20年の長きに渡って動き続けられることである。もしこのアプリケーションが東京メトロの公式サービスになったとしよう。都心の大動脈たる鉄道を司る会社ゆえ、何十万人、あるいは何百万人ものユーザーを抱えることになり、極めて公共性の高いサービスになるだろう。現在の鉄道経路検索サービスなどは、それをあてにして一日のスケジュールを決める人がいるように、本サービスもそのように生活の一部に溶け込む可能性がある。

もしそうなった時、突然依存している言語やライブラリー、あるいは OS に脆弱性が見つかり、アップデートパッチが出たら、即時の対応を迫られるだろう。しかし、本当にアップデートしても大丈夫なのだろうか。いくら脆弱性が解消されるとはいえ、アップデートの弊害でソフトウェアが動かなくなり、修正するまでサービス停止に追い込まれてしまったら？ それに生活を依存している何十万もの人々が路頭に迷い大混乱が起きてしまう。鉄道情報程度ならまだいいが、もしそれが運行システムだったり、金融システムだったり、医療システムだったりしたら……。しかし、POSIX 原理主義に基づいた作り方をしていれば、先も述べたようにほぼ安心してアップデートができるし、何なら OS ごと乗り換えることだって簡単にできる。心配は皆無とまでは言わないが、圧倒的に素早い対応ができるだろう。もともとこのコンテストはロンドンオリンピックに際して行われた同様のコンテストに倣って企画されたいが、メトロパイパーは Web API の公開さえ続く限り 2020 年の東京オリンピックまで何の苦労もなく動かし続けられる自信がある。果たして 2020 年まで動き続ける作品は他にどれだけあるだろうか。

■依存ソフトの仕様変更はアンコントロール このようなメトロパイパーの特長を説明すると、「でも新路線ができたりして路線仕様が変わったら意味ないじゃん」と言われたりもする。確かに、丸ノ内線、千代田線、南北線、有楽町・新都心線などの分岐を持つ路線には個別対応するプログラムになった。

しかし、飽くまで公式サービスになった時の話として考えてみてもらいたい。路線が増えるなどといった路線の仕様変更は自社でコントロールできる話であるのに対し、依存言語等のバージョンアップはそのソフトウェア開発団体の一存で決まりこちらの会社の都合など聞いてくれはしない。大抵は「〇年×月までに移行してくださいとか、△△の機能が削除されます」といったアナウンスがなされるが、細かな機能の追加削除に関してはアナウンス漏れもあるだろうし、緊急時にはそれこそゼロデイでアップデートを迫られる。まさにアンコントロールだ。

例えば PHP は 5.2 から 5.3 へのアップデートによって比較的多くの機能が下位互換性が損なわれ、開発者に大きなインパクトをもたらした。このようにして依存ソフトウェアの変更に、我々プログラマー達は散々苦しめられてきた。この本が提唱する POSIX 原理主義プログラミングによって、少しでもデブロイや保守の苦勞から解放される人が増えれば幸いである。

POSIX 原理主義に対する誤解

POSIX 原理主義の説明をしていると、次のような反論を受けることがある。

いくら POSIX の範囲で独自コマンドを作っても、それを他人に提供すれば（＝他人の作った独自コマンドを使えば）、アンコントロールにバージョンアップがなされるので同じことではないか。

確かに、ここまでの説明ならばそう解釈されても仕方が無い。だが実際は、独自コマンドを使うにあたっては次のような条件が付く。

使い始めたら、自分の手足の如く、自分で扱い方を理解し、責任を持つこと。

「自分の手足の如く」というニュアンスを理解してもらいたい。自分の手足の血管がどうなっていて、骨がどうなっているといったことまで知っているわけではないが、例えば「人より関節が柔らかくて細いパイプの中にも拳を入れられる」といった特徴やクセを理解しているといった意味である。コマンドで言うなら「コード中身 1 行 1 行まで知り尽くしているわけではないが、その動作やクセは大体把握できており、使い方の工夫もトラブル対応もある程度自分で行える」という状態だ。

よって **POSIX 原理主義**においては、他人の作った独自コマンドであっても、それを手に入れたユーザーは自分の作ったコマンドとして扱わなければならないというルールを定めている。その為、本書で紹介している拙作の独自コマンドは全てパブリックドメイン（ライセンス放棄状態）で提供している。

20 年後など、誰も責任とってはくれない

ソフトウェアに限った話でもないが、悲劇が生まれる原因は、他人が作ったものに過度に依存しすぎていることにあると私は思う。技術が高度化した現代においては、道具のブラックボックス化が進み、特にその傾向強い。また、利用者も道具の特性まで興味を示す余裕がなくなってしまった。POSIX 原理主義は「原理主義」という名に相応しく、そういった流れに抗う主義である。

そもそも、20 年も先のことを保証してくれるものなど国の社会保障くらいである*³。「20 年耐えうる性能です」とは主張しても、「20 年後も私が責任を持ちます」などという人はおそらく居ない。20 年後にも生きて同じ職業を続けている保証など無いのだから。それは本書の筆者とて同じである。

それならば 20 年間本当に動き続ける保証のあるソフトウェアを手に入れるにはどうすればいいのか。答えは、他人に頼らず自分で保守できる知恵を身に付けること、だ。

本書の効果的な活用方法

最後に、レシピ集である本書の具体的な使いこなし方を記しておく。

シェルスクリプトの基礎については他書を併読する

本書はレシピ集である。つまり「目の前の課題に対し、既存の文法やコマンドをどのように駆使すればそれが解決できるか」を紹介する本である。

従って、「既存の文法」や「コマンド」といったものを予め知っていなければ、本書のレシピを理解し、活用することは難しい。できればそれら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、文法やコマンドを解説しているシェルスクリプトの教本としては既に多数の良書が存在する。

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、書店のコンピューター書売り場（UNIX 関係）で良書を探るか、Web 上でシェル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。

POSIX 規格を見ながら開発する

本書を参考にしながら、時空を超えた可搬性を持ち合わせたプログラムを実際開発するには、POSIX 規格が具体的にどのような内容になっているのかを常に確信しながら進めることが必要不可欠である。

*³ ちなみにこの一文は、確実性が疑問視されている現状の社会保障制度に対する皮肉である。

Web ページの場所

POSIX の内容は、誰でも Web 上で簡単に閲覧することができる。実際に POSIX 原理主義に基づいてアプリケーションを制作する際には、本書と併せ、POSIX の原典を見ながら進めるべきである。

検索エンジン等で“opengroup POSIX”という 2 つのワードで検索すれば、“Posix - The Open Group”という名のページがすぐに見つかるだろう。執筆時の最新版は 2013 年版であり、

<http://pubs.opengroup.org/onlinepubs/9699919799/>

という URL である。

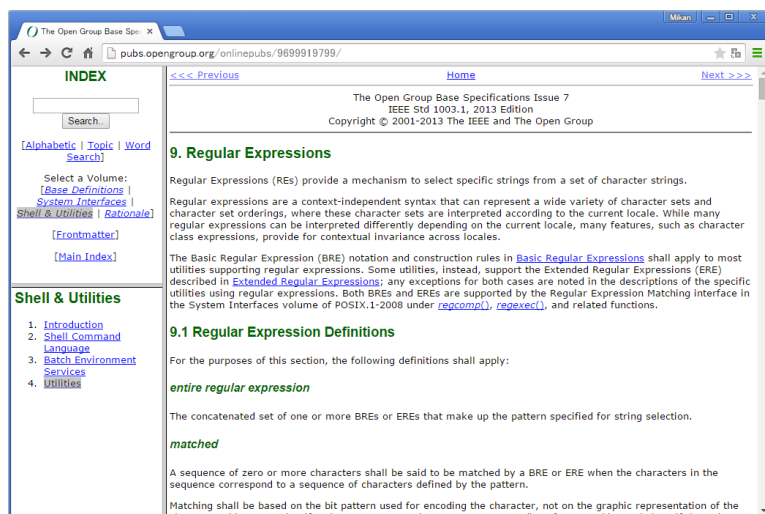


図 3 POSIX 規格の原典 “Posix - The Open Group”

コマンドマニュアル

最もお世話になる機会が多いと思われるコマンド仕様のページは、左上のメニューから“Shell & Utilities”を選び、続いて左下に現れる“Utilities”を選べば出てくる。

尚、文章は全て英語である。もし英語が苦手という人は、“FreeBSD 日本語 man”や“Linux JM”（Linux の日本語マニュアル）で検索される日本語マニュアルのページと併読するのがよいだろう。これらの man で書かれている内容は殆ど POSIX のページに書かれている内容のスーパーセット（上位互換）になっているので、「日本語で書かれているこのコマンドオプションは POSIX にも載っている」などと見比べながら確認するとよい。

用いるのは Bourne「系」シェルスクリプトである

ここまで来て言うまでもないが、開発で使用する言語はシェルスクリプト、しかも Bourne シェルのシェルスクリプトである。Perl、PHP、Ruby、Python その他の Lightweight Language と呼ばれる言語は一切用いない。理由は POSIX で規定されているのが Bourne シェルだからである。

ただし、bash、dash、ksh、zsh 等の（C シェルではない）**Bourne** 系のシェルを使ってはならないということではない。これらは全て Bourne シェルの上位互換であり単にそれらのシェルが独自拡張した機能を使わないというだけだ。同じ理由で、GNU AWK 等のコマンドもちろん使って構わない。

コマンドセット “Open usp Tukubai” を活用する

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、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*4。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 Open usp Tukubai（無償版）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 氏によって、これらのコマンドを POSIX の範囲で動くシェルスクリプトに移植する作業が行われている*5。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。

また、この場を借りて 321516 氏、そしてオリジナルを公開している USP 研究所に感謝したい。

制約を理解する

道具を使いこなすコツの一つは、その道具の性質および、できることできないことを正しく理解することである。本書についても若干の制約事項があるので述べておく。

POSIX 準拠であればいいとはいえない例外

本書は、基本的には **POSIX** 規格に準拠する内容であるように注意を払っている。より具体的には、“IEEE Std 1003.1” という規格への準拠である。これはもちろん、「時空を超えた互換性」等の利点を引き出すためである。

だが、「基本的に」という断りを付けた。これには理由がある。歴史上、POSIX は各種 UNIX 系 OS で共通している仕様を抜き出す形でまとめられた規格であるが、相反する仕様を持っていることが原因でどちらかの使用を選ばざるを得なかったものが若干ある（tr コマンドの仕様など）。選ばれなかった側の UNIX 系 OS が現存しない（＝サポート終了している）のであればよいが、現行品もあり、そのような OS を無視しては本末転倒である。

そこで、いくら POSIX 規格で明記されていても、本書では全てを推奨しているわけではない。ただしそのようなものについては個別に明記している。

動作試験環境

紹介しているレシピは全ての UNIX 系 OS 上で動くことを目標にしているので、本来であれば全ての UNIX 系 OS 上でレシピを試食しなければならない。しかし現実的にそれは不可能であるため、もしかするとお使いの環境によってはご賞味頂けないレシピがあるかもしれないことをご了承ください。レシピを試食した主な OS は、FreeBSD 9～10、CentOS 5～7、AIX、Raspbian などである。

*4 公式サイト→ <https://uec.usp-lab.com/TUKUBAI/CGI/TUKUBAI.CGI?POMPA=ABOUT>。尚、高いパフォーマンスを発揮する有償版 “usp Tukubai” というものも存在する。

*5 GitHub 上で公開中→ <https://github.com/ShellSchoccar-jpn/Open-usp-Tukubai/tree/master/COMMANDS.SH>

おことわり

細心（最新）の注意を払ってはいるもののその他にも、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ツコミなどお寄せ頂きたい。

`richmikan@richlab.org`

目次

まえがき	ii
第 1 章　ちょっとうれしいレシピ	1
レシピ 1.1　ヒストリーを残さずログアウト	1
レシピ 1.2　sed による改行文字への置換を、綺麗に書く	2
レシピ 1.3　grep に対する fgrep のような素直な sed	3
レシピ 1.4　mkfifo コマンドの活用	4
レシピ 1.5　一時ファイルを作らずファイルを更新する	7
レシピ 1.6　テキストデータの最後の行を消す	11
レシピ 1.7　改行無し終端テキストを扱う	12
レシピ 1.8　IP アドレスを調べる (IPv6 も)	15
レシピ 1.9　YYYYMMDDhhmmss を年月日時分秒に簡単に分離する	17
レシピ 1.10　祝日を取得する	18
レシピ 1.11　ブラックリストの 100 件を 1 万件の名簿から除去する	20
第 2 章　利用者の陰に潜む、様々な落とし穴	25
レシピ 2.1　【緊急】false コマンドの深刻な不具合	25
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	26
レシピ 2.3　全角文字に対する正規表現の扱い	29
レシピ 2.4　sort コマンドの基本と応用とワナ	30
レシピ 2.5　sed の N コマンドの動きが何かおかしい	37
レシピ 2.6　標準入力以外から AWK に正しく文字列を渡す	39
レシピ 2.7　AWK の連想配列が読み込むだけで変わるワナ	41
レシピ 2.8　while read で文字列が正しく渡せない	42
レシピ 2.9　あなたはいくつ問題点を見つけられるか!?	43
第 3 章　POSIX 原理主義テクニック	47
レシピ 3.1　PIPESTATUS さようなら	47
レシピ 3.2　Apache の combined 形式ログを扱いやすくする	51
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	53
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	56
レシピ 3.5　CSV ファイルを読み込む	59
レシピ 3.6　JSON ファイルを読み込む	61
レシピ 3.7　XML、HTML ファイルを読み込む	64

レシピ 3.8 全角・半角文字の相互変換	66
レシピ 3.9 ひらがな・カタカナの相互変換	69
レシピ 3.10 バイナリーデータを扱う	71
レシピ 3.11 ロック（排他・共有）とセマフォ	74
レシピ 3.12 デバッグってどうやってるの？	83
第 4 章 POSIX 原理主義テクニック – Web 編	89
レシピ 4.1 URL デコードする	89
レシピ 4.2 URL エンコードする	91
レシピ 4.3 Base64 エンコード・デコードする	92
レシピ 4.4 CGI 変数の取得（GET メソッド編）	93
レシピ 4.5 CGI 変数の取得（POST メソッド編）	95
レシピ 4.6 Web ブラウザーからのファイルアップロード	97
レシピ 4.7 Ajax で画面更新したい	99
レシピ 4.8 シェルスクリプトでメール送信	103
レシピ 4.9 メールマガジンを送る	108
レシピ 4.10 HTML テーブルを簡単綺麗に生成する	111
レシピ 4.11 シェルスクリプトおばさんの手づくり Cookie（読み取り編）	117
レシピ 4.12 シェルスクリプトおばさんの手づくり Cookie（書き込み編）	118
レシピ 4.13 シェルスクリプトによる HTTP セッション管理	121
第 5 章 どの環境でも使えるシェルスクリプトを書く	126
レシピ 5.1 環境変数等の初期化	126
レシピ 5.2 シェル変数	127
レシピ 5.3 スコープ	128
レシピ 5.4 正規表現	128
レシピ 5.5 文字クラス	129
レシピ 5.6 乱数	129
レシピ 5.7 ロケール	130
レシピ 5.8 \$((式))	132
レシピ 5.9 case 文	132
レシピ 5.10 if 文	132
レシピ 5.11 local 修飾子	133
レシピ 5.12 PIPESTATUS 変数	133
レシピ 5.13 “[” コマンド	140
レシピ 5.14 AWK コマンド	140
レシピ 5.15 date コマンド	143
レシピ 5.16 du コマンド	143
レシピ 5.17 echo コマンド	145
レシピ 5.18 exec コマンド	145
レシピ 5.19 fold コマンド	146
レシピ 5.20 grep コマンド	146

レシピ 5.21 head コマンド	147
レシピ 5.22 ifconfig コマンド	148
レシピ 5.23 kill コマンド	148
レシピ 5.24 mktemp コマンド	149
レシピ 5.25 nl コマンド	150
レシピ 5.26 printf コマンド	151
レシピ 5.27 ps コマンド	152
レシピ 5.28 readlink コマンド	153
レシピ 5.29 sed コマンド	153
レシピ 5.30 sort コマンド	154
レシピ 5.31 tac コマンド・tail コマンド “-r” オプションによる逆順出力	154
レシピ 5.32 test (“[”) コマンド	155
レシピ 5.33 tr コマンド	156
レシピ 5.34 trap コマンド	157
レシピ 5.35 which コマンド	157
レシピ 5.36 xargs コマンド	157
レシピ 5.37 zcat コマンド	160
 第 6 章 レシピを駆使した調理例	 161
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	161
 あとがき	 172

第 1 章

ちょっとうれしいレシピ

本章では、普段の作業の辛いところに手が届くような、ちょっとうれしいレシピを紹介する。すごくうれしいレシピは、後ろの章で紹介するので楽しみに。

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ください。

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、今ログイン中のシェルの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、置換後の文字列の中で改行を入れたい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ふ*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の中で使用すればよい。

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。

取り除かれなくするには、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。

レシビ 1.3 grep に対する fgrep のような素直な sed

問題

テキストファイルの中に自分で定義したマクロ文字列を起き、それをシェル変数に入っている文字列で置換したい。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' "$to" |
  sed 's/¥([^\$]/¥¥1/g' | # ・後方参照として意味を持つメタ文字をエスケープ
  sed 's/¥/¥¥/' | # ・文字列中の改行コードを
  sed 's/¥¥$//') # エスケープ

# あとは普通に sed に掛ければよい
cat template.txt | sed "s/$fr/$to/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。まあ、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）があるとする。###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で置換したいと思った時、

```
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？」と思うかもしれないが、sed はこれで大丈夫。なぜなら sed は BRE（拡張正規表現）にしか対応していないからだ*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 →??章第二部（正規表現）参照

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だろうかと思って、ターミナル A 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  
  
# 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} 読み込み始めはぼんやり表示され、データが読み進められると次第にクッキリ表示される JPEG ファイルである。

■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 一時ファイルを作らずファイルを更新する

問題

sedやnkfなど一部のコマンドでは、一時ファイルを使わずに内容を直接上書きする機能があるが、他のコマンドやシェルスクリプトではできないのか？いちいち一時ファイルに書き出してから元のファイルに再び書き戻すのは面倒だ。

回答

できる（大抵のファイルは）。ただしできないファイルもあるので、まずは次で可否を確かめること。

ハードリンクが他に存在するファイル

→できない。

所有者等が他人のファイル

→できない。

シンボリックリンク

→実体を探し、それが上記に該当しないファイルならばできる。

ACL 付ファイル

→ ACL 情報を保存し、復元すればできる。

デフォルトと違うパーミッションを持つファイル

→パーミッション情報を保存し、復元すればできる。

上記に該当しないファイル

→できる。

確認の結果、できるファイルであったとする。今、対象ファイルのパスがシェル変数`$file`に入っていて、更新のために通したいコマンドが `CMD1 | CMD2 | ...` だったならば、次のように記述すればよい。これで一時ファイル無しの上書き更新ができる。

```
(rm "$file" && CMD1 | CMD2 | ... >"$file") <"$file"
```

シンボリックリンクであった場合に実体を探す方法

シンボリックリンクであったらリンク元である実体ファイルを探し、それに対して前記のコードを実行しなければならない。もしシンボリックリンクに対して実行してしまうと、元のファイルは更新されずに残り、シンボリックリンクは更新された内容で実体化してしまう。

POSIX の範囲を超えるのがアリなら、次のように `readlink` コマンドを使えば実体のパスを簡単に得ることができる。

```
file=$(readlink -f "対象シンボリックリンクへのパス")
```

“-f” オプションは、リンク先がまたリンクであった場合に再帰的に実体を探すためのものである。これでシェル変数`$file`に実体が（存在すれば）代入されるので、この後先程と同様に `(rm "$file" && CMD1 ...` を実行すればよい。

だが、本書は POSIX 原理主義者のためのものであるからもちろんそれで済ませはしない。前記の `readlink -f` に相当するコードを POSIX の範囲で書くようになる。

■readlink -f 相当を、POSIX の範囲で実装する

```

while ;; do
# 1) ls コマンドで、属性とファイルサイズを取ってくる
s=$(ls -adl "$file" 2>/dev/null) || {
    printf '%s: cannot open "%s" (Permission denied)%n' "${0##*/}" "$file" 1>&2
    exit 1
}
# 2) ls が返した文字列を要素毎に分割
set -- $s # 属性は$1に格納、ファイルサイズは$5に格納される
# 3) 実ファイルを見つけるためのループ
#   ・そもそも見つからなければエラー終了
#   ・通常ファイルならそのファイル名でループ終了
#   ・リンクならリンク元のパスを調べて再度ループ
#   ・それ以外のファイルならエラー終了
case "$1" in
    '') printf '%s: %s: No such file or directory\n' "${0##*/}" "$file" 1>&2
        exit 1
        ;;
    -*) break
        ;;
    l*) s=$(printf '%s' "$file" |
        sed 's/¥([[] .¥*/[[] ¥)/¥¥¥1/g' |
        sed 's/^¥^/¥¥^/' |
        sed 's/¥$$$/¥¥$/' )
        srcfile=$(file "$file" |
            sed 's/^ .¥{"$s"}¥: symbolic link to //' |
            sed 's/^¥(. *¥)"'""'¥/¥1/' )
        case "$srcfile" in
            /*) file=$srcfile ;; # 返してくることは
            *) file="$file%/*)/$srcfile";; # POSIX で保証されている
        esac
        continue
        ;;
    *) printf '%s\n' "${0##*/}: ¥'$file' is not a regular file." 1>&2
        exit 1
        ;;
esac
done

```

先程と同じくシェル変数\$*file* に実体が（存在すれば）代入されるので、あとは同様である。

ACL 付やパーミッションが変更されているファイルへの対応

これらの場合は、上書き更新を実施する前にそれらの情報を保存しておかなければならない。次のコードはパーミッションと ACL を維持するための例だ。

■元のパーミッションと ACL も維持しつつ上書き更新する

```
file=対象ファイルのパス # シンボリックリンクなら実体を探しておくこと

# パーミッションと、ACL 情報（あれば）の保存
perm=$(ls -adl "$file" | awk '{print substr($0,1,11)}')
case "$perm" in
  [~-]*) echo "Not a regular file" 1>&2;exit 1;;
  *'+') acl=$(getfacl "$file")      ;;
  *)    acl=''                      ;;
esac

# パーミッションを chmod で使える形式（4 桁ゼロ埋め 8 進数）に変換
perm=$(echo "$perm" |
  sed 's/.[0-7][0-7][0-7][0-7]/[0-7][0-7][0-7][0-7]/' |
  awk '{gsub(/[/[x-]/,"0",$1);gsub(/[/[0]/,"1",$1);print $1 $2;}' |
  tr 'STrwxst-' '00111110' |
  xargs printf 'ibase=2;%s\n' |
  bc |
  xargs printf '%04o' )

# 一時ファイルを作らず上書き更新
(rm "$file" && CMD1 | CMD2 | ... > "file") < "$file"

# パーミッション・ACL（あれば）の復元
chmod $perm "$file"
case "$acl" in '' ) ;; * ) printf '%s' "$acl" | setfacl -M - "$file";; esac
```

ls コマンドの“-l”オプションが第1列に出力する文字列を見ている。それに基づいてパーミッション文字列をchmod コマンド用に8進数化し、保存・復元すると共に、ACL 情報の有無を確認し、あればそれも保存・復元している。

解説

一時ファイル無しで上書きするトリック

そもそも、(rm "\$file" && CMD1 ... というコードを書くとなぜ一時ファイル無しで上書きができるのだろうか。

UNIXにおいて、rm コマンド等による削除（unlink）は、ファイルの実体を消すのではなく inode と呼ばれる見出しを消すだけであることは御存知のとおり。つまり、ファイルが直ちに消滅するわけではない。

もし、誰かがファイルをオープンしている途中に削除したらどうなるかというと、以後は誰もファイルを二度とオープンできなくなるものの、そのファイルを既にオープンしているプロセスはクローズするまで使い続けることができる。つまり、ファイルをオープンして中身を読み出している間にそのファイルを削除をしてしまっても、読み出しは最後まで行えるのだ。そこでさかさず読み出されたデータを受け取って好きな加工を施した上で、同名（2代目）のファイルを新規作成する。

同名のファイルなど作れないように思うが、初代のファイル実体は既に inode を失っているために無名である。よって2代目ファイルを全く同じ名前で作成可能なのである。

リンク付・所有者が他人のファイルが不可な理由

先に、ハードリンク付・所有者が他人であるかどうか確認する方法を記すが、それらはどちらも `ls` コマンドの “`-l`” オプションを使えばわかる。

例えば、`ls -l` を実行した結果次のように表示されたとする。

```
$ ls -l readme.txt
-rw-rw-r--+ 3 mikan      staff   3513 Jun  1 18:43 readme.txt
$
```

ここで、第 2 列の数字は、自分を含めて同じ実体を共有しているハードリンクの数である。これが 1 でないと本レシピは使えない。理由は、共有している他のハードリンクの名前を知る術がなく、本レシピの方法で一旦削除してしまうとリンクし直せなくなってしまうからである。

一方、第 3 列、第 4 列にはファイルの所有者と所有グループが表示されているが、これらが自分でなければ本レシピは使えない。理由は、一般ユーザー権限では所有者と所有グループを他人のものにすることができず、元の状態に戻せないからである。

レシピ 1.6 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
.
MAIL
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
↓
← 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したら面倒臭い。

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' ↵
やあ皆さん、私の研究室へようこそ。
以上
.
MAIL
やあ皆さん、私の研究室へようこそ。
$
```

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

レシピ 1.7 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。だが、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 コマンドをフィルターとして使った場合には、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、ファイル終端に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は、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具体的などうやればいいのか。

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 (出力行区切り文字) 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 1.8 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。

■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¥.'            | | # loopback 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。

^{*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。

(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

方法その2は、シェル関数化してしまうというものだ。あちこちで使い回したい場合はこちらの方がよいだろう。シェル関数化したらそれを\$(~) で囲めば、方法その1と同じく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ういだろう。

^{*6} AIX など、ifconfig コマンドが/etc に置いてある OS なんてのがあつたのだ。

レシピ 1.9 YYYYMMDDhhmmss を年月日時分秒に簡単に分離する

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 で各列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。

```
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、先頭の 2 個（計 4 桁）だけ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 1.10 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 Web API に問い合わせさせて教えてもらう。

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（2015/06/01 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/{
    rec++;
    }
    match($0,/DTSTART.*DATE:/{
        print rec,1,substr($0,RLENGTH+1); # DTSTART 行は日付であるから
    }
    match($0,/SUMMARY:/{
        s=substr($0,RLENGTH+1);
        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);}
    $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 2 月 1 日、国立天文台から翌年の月日が発表されることで決まることになっているからだ。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 Web API 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、それらが取められているタグ (“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

^{*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>

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 Web API の URL は 2015/06/01 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

参照

→ RFC 5545 文書^{*9}

レシピ 1.11 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。

^{*9} <https://tools.ietf.org/html/rfc5545>

■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 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。

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
$
```

両方のファイルができたなら、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 > cleanedmembers.txt
$ wc -l cleanedmembers.txt
  9988 cleanedmembers.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 (del_blmembers.sh) で記した 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。

```
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。

^{*10} 厳密には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区切り文字である半角スペースが連続した箇所ができることになる。だがそれではわかりにくい。

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、join コマンド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。例えば人名列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列区切り文字をしっかりと定義しておくこと。

参照

→レシピ 2.4 (sort コマンドの基本と応用とワナ)

→レシピ 5.7 (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ど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ホールがあると聞いたが、一体どういう事か？

回答

2014 年、コンピューターセキュリティ史上一、二を争うであろうとても深刻なセキュリティ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 コマンドの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る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。

確かに前述のと通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勞と響感を買いたくないし、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 "HOGESPIPE"
$ { cat <"HOGESPIPE" >/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 | grep -E '$$'|'cat' | grep -Ev 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からだ。従って半角英数字だけを置換対象にしたいのであれば、環境変数を C ロケールにするか無効にする。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'
*** MSX2 MSX2+
                                     ←C ロケールにする
$ echo 'MSX MSX2 MSX2+' | env -i sed 's/[:alnum:]/*/g'
*** MSX2 MSX2+
                                     ←環境変数を無効化
$ echo 'MSX MSX2 MSX2+' | sed 's/[A-Za-z0-9]/*/g'
*** MSX2 MSX2+
                                     ←明確に半角英数字を指定
$
```

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、半角文字とそれに対応する全角文字を同一視する。知っていれば便利だろうが、知らずにそう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 5.7（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。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 <<EXAMPLE | sort
> perl
> ruby
> Perl
> Ruby
> EXAMPLE
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 <<EXAMPLE | sort -f
> perl
> ruby
> Perl
> Ruby
> EXAMPLE
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 <<EXAMPLE | sort -n
> 2
> 10
> -3
> 1
> EXAMPLE
-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 <<EXAMPLE | sort -g
> +6.02e+23
> 1.602e-19
> -928.476e-26
> EXAMPLE
-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 <<EXAMPLE | 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 ↵
> EXAMPLE ↵
+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)…駅データ

次のように、とある鉄道駅の開業年、快速停車の有無、駅名、ふりがなの 4 列から構成されるスペース区切りのデータ (sample1.txt) があったとしよう。

■ sample1.txt

```
1908 停車 相原 あいはら
1957 通過 矢部 やべ
1979 通過 成瀬 なるせ
1926 停車 菊名 きくな
1964 停車 新横浜 しんよこはま
1947 通過 大口 おおぐち
1908 停車 町田 まちだ
```

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「50 音順にソートせよ」という要請を受けたとする。ふりがなが 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 4 列目にある。こういう時は、-k 4,4 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 4,4 sample1.txt ↵
1908 停車 相原 あいはら
1947 通過 大口 おおぐち
1926 停車 菊名 きくな
1964 停車 新横浜 しんよこはま
1979 通過 成瀬 なるせ
1908 停車 町田 まちだ
1957 通過 矢部 やべ
$
```

なぜ2回書くのかについてであるが、入門段階ではとりあえず「そういうものだ」と思って覚えておけばよい^{*4}。

さて次に「50音順で降順にソートせよ」という要請を受けたとする。先程は昇順にソートしたが、降順にしたい場合はどうするか。答えは`-k 4r,4`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思っておけばよい。

```
$ sort -k 4r,4 sample1.txt ↵
1957 通過 矢部 やべ
1908 停車 町田 まちだ
1979 通過 成瀬 なるせ
1964 停車 新横浜 しんよこはま
1926 停車 菊名 きくな
1947 通過 大口 おおぐち
1908 停車 相原 あいはら
$
```

ちなみに、もし読み方が半角アルファベットで記述されていて、それを辞書順に並べたかったとするなら、`-k 4fr,4`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「快速停車の有無→開業年の新しい順→駅名の50音順でソートせよ」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 4,4`だ。

^{*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。

```
$ sort -k 2,2 -k 1nr,1 -k 4,4 sample1.txt ↵
1979 通過 成瀬 なるせ
1957 通過 矢部 やべ
1947 通過 大口 おおぐち
1964 停車 新横浜 しんよこはま
1926 停車 菊名 きくな
1908 停車 相原 あいはら
1908 停車 町田 まちだ
$
```

「通過」と「停車」を昇順にすると前者が先に来るのは、一文字目の「通」と「停」音読みした場合の名前順で前者の方が先だからである。また、開業年の降順ソート（`-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。（そんな駅あるのか!?!）

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。誰もが持つ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、これでは正しくソートできないので `sort` コマンドの直前には `grep -v '^#'` を挿み、この行を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。

問題は列区切り文字だ。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. 列区切り文字に潜むワナ 2 つ

おまじかねのワナ編。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。

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

理由は、列区切りルールがとても特殊であることに起因する。なんと `sort` コマンドはデフォルトでは、空白類（スペースやタブ）でない文字から空白類への切り替わり位置で列を区切り、しかも区切った文字列の先頭にその空白類があるものと見なす。つまり上記テキストの各列は、次の表の通りに解釈される。

	第1列	第2列	第3列
1行目	“1”	“ B”	“ _”
2行目	“10”	“ A”	“ _”

なぜこのようなルールにされたのかは全くの謎だが、「それならば」と-t オプションを用い、列区切り文字は半角スペースである（-t ' '）と指定してもうまいかない。この場合、対象となるテキストデータ中で半角スペースが複数個連続していると、その間に空文字の列があると見なされてしまうからだ。従って上記テキストは、1行目が第1列“1”の次に空文字の第2列があって4列から成ると解釈されてしまう。（次の表の通り）

	第1列	第2列	第3列	第4列
1行目	“1”	“ ”	“B”	“_”
2行目	“10”	“A”	“_”	“ ”

■どうすればいいのか 結局のところ sort コマンドに正しくソートさせるには、列と列を区切る複数のスペースを1個にしなければならない。だから、先ほど例示したテキストを正し順番にソートしたければ下記のように修正する。

```
$ 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 -
$
```

まあ当然、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

```
近江 舞子 おうみ まいこ
下神 明 しもがみ あきら
飯山 満 いいやま みつる
```

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。ふりがなが第

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
近江  舞子 おうみ  まいこ
飯山  満   いいやま みつる
下神  明   しもがみ あきら
$
```

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` オプションで列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
飯山  満   いいやま みつる
近江  舞子 おうみ  まいこ
下神  明   しもがみ あきら
$
```

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 1.11 (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、2014 年 11 月公開の FreeBSD 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、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（2015 年 6 月）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い動きをしていたら、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 "newline" 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、全ての文字が一切エスケープずに伝わる。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 の連想配列が読み込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)";
    split(str, ar);
    print "***DEBUG*** array#3:", ar[3];
    if ((1 in ar) && (2 in ar) && (3 in ar)) {
        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";
        exit;
    }
}'
```

← 1) 本来あるべき第三列"data(3/3)"が無い

← 3) ここにデバッグ用コードを入れたら
エラー終了しなくなりました!

← 2) 冒頭の問題により
ここでエラー終了してしまっていた

回答

これはバグではない。存在しない配列要素を読み込むと、その時点で空の要素を生成するという AWK の仕様な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、空文字を値としてその要素を勝手に作成してしまう。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ください。

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済みである。

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も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 つの 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、全部見つけれられるかな? :-)

:
:
:

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。そこで、環境変数 PATH がいじられていても影響を受けないように、次の行をシェルスクリプトの冒頭に追加する。

■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 (I) 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

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、絶対パス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

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て誤動作してしまう。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。そうならないように 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、処理を中断するようにすべきである。

■指定されたディレクト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れられない。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 -la |
```

7. ファイル名がスペースを含んでいる

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。よって両方ともファイル名を示しているシェル変数を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 3 と同じ理屈だ。そのうえもし、先の指摘 7 の対策コードも下記に記す対策コードもなく、“-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？」と言っているそのアナタ。POSIX を見くびるのはこの章を読んでからにしておうか。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。PIPESTATUS 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ほしい。

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="$l ¥"¥${$k}¥" # ←修正箇所はここ（「解説」を参照）
    fi
    k=$((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、シェル自身が自分への命令とし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* はコマンドの順番で、最初は 1）に格納されている。

解説

このシェル関数とはもと 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（先ほど示したコードに付けた 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 範囲外であるため、使えなければ外しつつ、関数の外で同名の変数を使わないように気を付けること。

^{*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。

実際に試してみる

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よう。

- 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。

シェル関数を使わずにやるには

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ろなる。

```
$ 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` コマンドに `-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 つの列とされている。それゆえ、アクセス日時の列、User-Agent の列など、任意の列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。これらを通すと、各列内の空白文字がアンダースコアに置換され、列区切りとしての空白だけが残るので、以後 AWK など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 '%Yn'); 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' "$LF"/g' |
sed 's/^¥[" []/s/[[[:blank:]]/¥"¥c"/g' |
tr -d '¥n' |
tr "$RS" '¥n'
```

このシェルスクリプトを通した後、日時列が欲しければ `awk '{print $4}'`、同様に HTTP リクエストパラメーター列なら `awk '{print $5}'`、User-Agent 列なら `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"
```

この中の User-Agent 列 ("`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 "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"
$
```

`apacomb_norm.sh` 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、先ほど行を独立させた区間なので、これらの行にある空白を空白でない文字列 (今回の例では `"_"`) 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/^[" []/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ほしい。

<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う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

日常使っている日時から 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ただけなので簡単だ。

■ 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' の出力文字列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 UNIX 時間" |
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_calculated_year = 1970;          # 各年の元日は 1970/01/01 から何日後なのかを
    days_on_Jan1st_from_epoch[1970] = 0; # ←記憶しておくための変数
    Y = int(days_from_epoch/365.2425)+1970+1;
    if (Y > max_calculated_year) {
        i = days_on_Jan1st_from_epoch[max_calculated_year];
        for (j=max_calculated_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_calculated_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ほしい。

<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。詳しくは以下のそれぞれのレシピを参照してもらいたい。

→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 4.12 (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 4.13 (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のか。

回答

比較したい日時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 (YYYYMMDDhhmmss1 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秒単位のタイムスタンプ比較が実現できることになる。

その計算はどうやるのかといえば、レシビ 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、これを逆変換して日常の時間に戻せばいいのだ。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 秒前より新しいファイル、等しいファイル、古いファイルを抽出するシェルスクリプトをそれぞれ紹介する。

■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 で規定されているオプションは **-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はあるから、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たことができる。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 つだったダブルクォーテーションを 2 個で表現しているから、1 行にあるダブルクォーテーションの数が偶数が奇数かの状態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

^{*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。

するまで、同一行と判断することができる。

仕様にに基づき、CSV パーサー “parsrc.sh” を実装する

ここから先は、先ほど紹介した parsrc.sh のソースコードを眺めながら読んでもらいたい。

■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 (SI:<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 (SO:<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 (行区切りを意味する RS:<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列番号を 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

Web API 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 “`¥t`” や “`¥n`” 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 `parsrj.sh` の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、私はなぜ再発明したのか。確かに、拙作のコマンドなら POSIX 原理主義に基づいているため、「どこでも動く」「10 年後も 20 年後も、たぶん動く」「コピー発デブロイ完了」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るので、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 を処理したいと思ったら `jq` コマンド、XML を処理したいと思ったら `xmllint` や `hxselect` (`html-xml-utils` というユーティリティの 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・ガンカーズの提唱する定理に

定理 1. 小さいものは美しい。

定理 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、部分抽出と思ったら `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

同じく定理の一つに

定理 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 xmlint 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 (スペース区切り) 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。UNIX コマンド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 哲学の定理だが、

定理 6. ソフトウェアは「てこ」。最小の労力で最大の効果を得よ。

定理 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。

というものがある。その定理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rinaさい」と書かれているそう。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

Web API 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 のパースももちろん可能だ。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。HTML は文法が間違っている 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、JSON の時と同様に階層構造をとるファイルパスのようにして値の格納場所を 1 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 `parsrx.sh` の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。これはシェルスクリプト開発者向けコマンドセット Open usp Tukubain にある同名コマンドを、POSIX 原理主義に基づいたシェルスクリプトで書き直した互換コマンドである。これをダウンロード^{*11}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
M o g a m i      yes   no
Kaga          no    yes
fubuki        yes   yes
m u t s u        no    no
S h i m a k a z e 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 Shift 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（こちらは 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 Open usp Tukubai に存在する同名コマンドを POSIX 原理主義シェルスクリプトで書き直した互換コマンドが存在する。必要に応じてダウンロード^{*12}して用いてもらいたい。

^{*11} <https://github.com/ShellShoccar-jpn/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。

^{*12} <https://github.com/ShellShoccar-jpn/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。

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「°」が連なっていたら「ハ°」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次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 哲学の定理その 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)

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^{*13}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このレシピを参照すること。

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。これはレシピ 3.8（全角・半角文字の相互変換）で紹介した `han`、`zen` コマンドのインターフェースに似せる形で作られている。これをダウンロード^{*14}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。このように時に、次のようにして `hira2kata` を使うのである。

^{*13} MSX など半角ひらがなを持っているコンピュータはあるのだけど一般的ではない。

^{*14} <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   ゼカマン
$
```

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 列だけ変換せよという意味である。よって第 1 列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、列という概念無しに、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 列の「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 Shift 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（こちらは POSIX ではないが）を用いて予め UTF-8 に変換しておくこと。

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、POSIX の範囲内で han、zen コマンドのインターフェースに似せる形で作られたものである。あわせてダウンロード^{*15}しておくといだろう。

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抑えている。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^{*16}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。

参照

→レシピ 3.8（全角・半角の相互変換）

^{*15} <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。

^{*16} 『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が参照になる。

→ <http://orange-factory.com/sample/utf8/code3-e3.html>

レシピ 3.10 バイナリーデータを扱う

問題

バイナリーデータを AWK や read コマンドで読み込むとデータが壊れてしまう。バイナリデータをシェルスクリプトで扱うことはできないのか？

回答

頑張ればできないことはないが、基本的にはシェルスクリプトではバイナリデータを扱えない。AWK (GNU 版除く) の変数や、シェル変数 (zsh 除く) には、NULL 文字 < 0x00 > を格納することができないからだ。

■AWK(非 GNU 版) 変数に NULL 文字を読ませると

```
$ printf 'This is ¥000 a pen.¥n'| awk '{print $0;}'  
This is          ← NULL 終端扱いされて、a pen. が無かったことに  
$
```

■シェル変数 (zsh 以外) に NULL 文字を読ませる

```
$ str=$(printf 'abc¥000def')  
$ echo "$str"  
abcdef          ← NULL 文字以降が切れていないので成功したように見えるが  
$ echo ${#str}  
6               ← NULL 文字が無視されて 6 文字になっていた  
$
```

しかし「できない」終わらせてしまうとレシピにならないので、頑張って扱う方法を解説する。

解説

まず現状を整理しなければならない。「基本的には扱えない」とは言うものの、厳密には扱えるコマンドと扱えないコマンドがある。主なものを列挙すると次のとおりだ。

表 3.1 主要コマンドのバイナリデータ取り扱い可否

バイナリデータ取り扱い不可	バイナリデータ取り扱い可
awk bash cut echo grep sed sh sort xargs	cat dd head od printf tail tr wc

取り扱い可能なコマンドの中に od と printf があることに注目してもらいたい。od はバイナリーをテキストに変換するコマンドであり、printf はテキストからバイナリデータを出力するのに使えるコマンドである。よっ

て、バイナリデータをどうしてもシェルスクリプトで扱いたければ、**od** コマンドでテキスト化し、テキストデータで処理し、**printf** でバイナリーに戻すという方針をとればよい。

バイナリーデータ扱うシェルスクリプト

前述の方針に基づいて作った、バイナリーデータを扱うシェルスクリプトを記す。

データ加工パートと記した箇所には、標準入力から 1 行ずつ、16 進数表現された 1 バイト分のアスキーコードが到来する。これを加工して、16 進数表現された 1 バイト分のアスキーコードを標準出力に送ればよい。

■バイナリーデータを読んでそのまま書き出すシェルスクリプト

```

#!/bin/sh

# === バイナリ→テキスト変換パート =====
LF=$(printf '\n'); LF=${LF%_}      # 0) 準備
od -A n -t x1 -v -                | # 1) 16 進ダンプ (アドレス無し)
tr -Cd '0123456789abcdefABCDEF\n' | # 2) 空白はトル (でも改行コードは残すべし)
sed "s/./&${LF}/g"                 | # 3) 1 バイト 1 行に (しなくてもよいけど)
grep -v '^$'                        | # 4) sed で出来た空行をトル
#
# === データ加工パート =====
# (1 行毎に 16 進数表現された 1 バイト分のアスキーコードが得られるので
#   ここで煮るなり焼くなりいろいろやる)
#
# === テキスト→バイナリ変換パート =====
awk -v ARGMAX=$(getconf ARG_MAX) '
BEGIN{
    # 0) --- 定義 -----
    ORS    = "";                      # 引数の
    LF      = sprintf("\n");          # 最大許容文字列長は
    maxlen = int(ARGMAX/2) - length("printf "); # ← ARG_MAX の約半分とする
    arglen = 0;
    # 1) --- バイナリ変換用のハッシュテーブルを作る -----
    # 1-1) 全ての文字が素直に変換できるものとして一旦作る
    for (i=1; i<256; i++) {
        hex = sprintf("%02x",i);
        fmt[hex] = sprintf("%c",i);
        fmtl[hex] = 1;
    }
    # 1-2) printf で素直には変換できない文字をエスケープ表現で上書きする
    fmt["25"]="%%"; fmtl["25"]=2; # "%"
    fmt["5c"]="\\c"; fmtl["5c"]=4; # (back slash)
    fmt["00"]="\\0"; fmtl["00"]=5; # (null)
    fmt["0a"]="\\n"; fmtl["0a"]=3; # (Line Feed)
    fmt["0d"]="\\r"; fmtl["0d"]=3; # (Carriage Return)
    fmt["09"]="\\t"; fmtl["09"]=3; # (tab)
    fmt["0b"]="\\v"; fmtl["0b"]=3; # (Vertical Tab)
    fmt["0c"]="\\f"; fmtl["0c"]=3; # (Form Feed)
    fmt["20"]=" "; fmtl["20"]=5; # (space)
    fmt["22"]="\\"; fmtl["22"]=2; # (double quot)
    fmt["27"]="'"; fmtl["27"]=2; # (single quot)
    fmt["2d"]="\\055"; fmtl["2d"]=5; # "-"
}
{
    # 2) --- 出力文字列を printf フォーマット形式で生成 -----
    linelen = length($0);
    for (i=1; i<=linelen; i+=2) {
        if (arglen+4>maxlen) {print LF; arglen=0;}
        hex = substr($0, i, 2);
        print fmt[hex];
        arglen += fmtl[hex];
    }
}
END {
    # 3) --- 一部の xargs 実装への配慮で、最後に改行を付ける -

```

```

    if (NR>0) {print LF;}                                #
}                                                         #
,                                                         |
xargs -n 1 printf

```

コードの概要

たかがバイナリーデータの入出力だけでこれだけの行数になったのは、それなりにノウハウが詰まっているからであるが、それぞれを大ざっぱに解説する。

■バイナリーデータの取り込み こちらは比較的簡単である。od コマンドでダンプし、テキスト化すれば大方の作業は終わる。

ダンプすると通常アドレスが付くが、取り込みという作業では不要なので付けない。するとデータ本体である16進数文字と位置取りのスペース等が来るので、これを2文字ずつ改行すれば、1行1バイト分の16進数アスキーコード列になる。

■バイナリーデータの書き出し こちらは大変である。printf のフォーマット文字列部分に書き出したい文字列を指定すれば制御文字も含めて全ての文字を出力することができる。

とはいえ、1バイト毎に printf コマンドを起動するのはあまりにも非効率なので起動回数を抑えたい。そこで、printf に渡す前にアスキーコードから元の文字に変換しても差し支えないものを AWK で変換してしまう。ただし AWK や printf やシェル、それぞれで特殊な意味を持つ文字が多数あるので注意が必要だ。書き出し部分のコードが長い原因は主にこのせいである。

そして ARG_MAX (コマンド1行の最大文字列長) を見ながら printf のフォーマット文字列をできるだけ長く作り、これを xargs でループさせるというわけである。しかし ARG_MAX は、ギリギリまで使おうとするとなぜかエラーを起こす xargs 実装があるため、半分だけ使うようにしている。

参照

→レシピ 4.3 (Base64 エンコード・デコードする)

レシピ 3.11 ロック (排他・共有) とセマフォ

問題

プログラムを書いていると、

1. 複数プロセスから同時にファイルを読み書きされたくない (排他ロックがやりたい)
2. 同時読み込みはいいが、その間書き込みはされたくない (共有ロックがやりたい)
3. CPU コア数までしか同時にプロセスを立ち上げたくない (セマフォがやりたい)

ということがよくあるが、シェルスクリプトだとこれらができないので他言語に頼らざるを得ない。それでも POSIX 原理主義を貫けと言うのか？

回答

大丈夫、できる。確かに POSIX の範囲にはロックやセマフォを直接実現するコマンドが存在しない。しかし、1つのディレクトリーには同じ名前のファイルが作れないというファイルシステムの基本的性質を活用し、早い者勝ちでファイルを作れたプロセスにアクセス権を与えるというルールを設け、さらにいくつかのアイデアを凝らせば全て実現可能だ。（詳細は解説で述べる）

ただ、いくらできるといっても毎回書くに大変な量なので、一発でできるコマンドを作ったからこれを使うといい。まずは次のものをダウンロードしてパスを通しておく。

表 3.2 ロック・セマフォ関連コマンド

目的	コマンドのダウンロード場所
排他ロック	https://github.com/ShellShoccar-jpn/misc-tools/blob/master/pexlock
共有ロック	https://github.com/ShellShoccar-jpn/misc-tools/blob/master/pshlock
セマフォ	（上記の共有ロック用コマンドを用いる）
ロック・セマフォ解除	https://github.com/ShellShoccar-jpn/misc-tools/blob/master/punlock
解除漏れファイルの清掃	https://github.com/ShellShoccar-jpn/misc-tools/blob/master/pcllock

ロック・セマフォの実現にはファイルを活用すると述べたが、そのファイルを作るための「ロック管理ディレクトリー」を用意する。ここでは便宜的に /PATH/TO/LOCKDIR であるものとする。

排他ロック

例えば、ある会員データファイル（/PATH/TO/MEMBERS_DB）を編集するプログラムがあって、他のプロセスが編集・あるいは照会していない隙に安全に読み書きできるようにするには、次のようなコードを書けばよい。

```

:
:
# "members_db"というロック ID で排他ロック権を獲得する
# (成功するまで最大 10 秒待つ)
lockinst=$(pexlock -w 10 -d /PATH/TO/LOCKDIR members_db)
if [ $? -ne 0 ]; then
    echo '*** failed to lock "MEMBERS_DB"' 1>&2
    exit 1
fi

↑
排他ロック区間（ここで"/PATH/TO/MEMBERS_DB"を読み書きしてよい）
↓

# 排他ロック権を解放する（先程得たロックインスタンス"$lockinst"を指定する）
punlock "$lockinst"
:
:

```

このコードは、次に紹介する共有ロックプログラムと一緒に使うことができる（pexlock と pshlock は互いを尊重する）。

共有ロック

例えば、ある会員名簿ファイル（/PATH/TO/MEMBERS_DB）の内容を照会するプログラムがあって、他のプロセスが編集していない隙（読み出しているプロセスはいてもいい）に安全に読み出せるようにするには、次のようなコードを書けばよい。

```

        :
        :
# "members_db"というロック ID で共有ロック権を獲得する
# (成功するまで最大 10 秒待つ)
lockinst=$(pshlock -w 10 -d /PATH/TO/LOCKDIR stock_db)
if [ $? -ne 0 ]; then
    echo '*** failed to lock "STOCK_DB"' 1>&2
    exit 1
fi

↑
共有ロック区間 (ここで"/PATH/TO/MEMBERS_DB"を読み込んでよい)
↓

# 共有ロック権を解放する (先程得たロックインスタンス"$lockinst"を指定する)
punlock "$lockinst"
        :
        :

```

このコードは、先程紹介した排他ロックプログラムと一緒に使うことができる（pexlock と pshlock は互いを尊重する）。

セマフォ

例えば、CPU に高い負荷を掛けるプログラム（heavy_work.sh）があり、これを 8 コアの CPU を搭載しているホストで動かすため、同時起動を最大 8 プロセスに制限したい場合には、次のようなコードを書けばよい。

```

        :
        :
# "heavy_task"というロック ID で、最大同時アクセス数 8 のセマフォを取得する
# (成功するまで最大 10 秒待つ)
seminst=$(pshlock -n 8 -w 10 -d /PATH/TO/LOCKDIR heavy_task)
if [ $? -ne 0 ]; then
    echo '*** failed to get a semaphore for "heavy_task"' 1>&2
    exit 1
fi

# ここで、高負荷プログラムを実行する
heavy_work.sh

# 終わったらセマフォを返却する (先程得たセマフォインスタンス"$seminst"を指定する)
punlock "$lockinst"
        :
        :

```

ロック・セマフォ解除漏れへの対策

今紹介した各プログラムに強制終了の恐れがあったり、バグによって解除忘れがあると他のプログラムがロック権を獲得できなくなってしまう。そのような場合には、定期的にロック管理ディレクトリーをスキャンし、一

定の時間が経った古いロックファイル（かつ持ち主のプロセスが既に存在しない）を削除する必要がある。

これを行うためのコマンドが `pcllock` であり、例えば `crontab` に次のように登録して使う。

■前記のロック・セマフォプログラムを実行しているユーザーの `crontab` ファイル

```
      :  
      :  
# 5 分 (=300 秒) 以上経ち、かつ生成元プロセスが既にあるロックファイルを毎分監視  
* * * * * pcllock -l 300 -w 10 /PATH/TO/LOCKDIR
```

解説

シェルスクリプトでロックやセマフォを実現するための原理は、1つのディレクトリーには同じ名前のファイルが作れないという性質の活用であることは既に述べた。それを聞いて「なんとベタな方法なのか！」と落胆するかもしれないが、悔ることなかれ。もともと OS は、物理的に 1 台しかないディスクへのアクセス要求を捌くため、内部で必ず排他制御をやっている。従ってファイルによるロック管理とは、OS が備えている洗練された排他制御機構の活用に限らないのだ。

だが、それでは排他ロックの説明にしかっていない。共有ロック、セマフォまで含めて、どうやって実現しているのかを解説しよう。まずはこの後の説明のため、2つの基本ルールを確認しておく必要がある。

基本ルール 1. ロックファイルを作れた者勝ち

早い者勝ちでロックファイルを作るというルールの詳細は次のとおりとする。

- 早い者勝ちでファイル（ロックファイル）を作る。
- 「成功者はロック成功（アクセス権取得）」と取り決める。
- 失敗者は暫くしてから再度ロックファイル作成を試みる。
- 成功者は用事が済んだらロックファイル消す。

これを基本ルール 1 として制定する。

基本ルール 2. 一定の時間が経った古いロックファイルは消してよい

実際は、成功者がロックをしたまま消し忘れたり、何らかの理由で異常終了してしまうとアクセス権が紛失されてしまうという問題がある。仕方が無いので、ロックファイルのタイムスタンプを確認し、一定以上古いロックファイルは消してよいという基本ルール 2 を制定する。ただ、可能ならば生成元のプロセスが既に存在しないことも確認する方が親切だ。

ただし、重大な注意点が 1 つある。

一定以上古いロックファイルを消すという役割は、1つのロック管理ディレクトリーに対して 1つのプロセスにしか与えてはならない。

ということだ。もし、あるプロセス A が一定以上古いロックファイル検出し、今からそれを消して新たに作り直すとしている時、プロセス B が同じロックファイルを古いと判断して削除し、新規作成まで済ませてしまったらどうなるか。プロセス A はこの後、プロセス B の作ったロックファイルを誤って消してしまうことになる。これは、古いファイルの検出、削除、作り直しという操作が、アトミックに（単一操作で）できないという制約に起因する。よって、古いロックファイルの削除役は一人ではなければならないのである。

尚、ここで出てきた「一定以上古いファイルを検出する」という処理が POSIX の範囲では面倒なのだが、こ

れに関してはレシピ 3.4 (find コマンドで秒単位にタイムスタンプ比較をする) を利用すれば解決できるので、ここでは割愛する。

以上を踏まえ、各ロックを実現するアイデアをまとめる。

その 1. 排他ロックはどうやるか

排他ロックとは、誰にも邪魔されない唯一のアクセス権を獲得するためのロックだ。ファイルを読み書きする場合などに用いる。

ファイルを用いて排他ロックを実現する方法というのは実はよく知られており、単純である。ロック管理用ディレクトリの中でロックファイルを作ればよいわけだが、既存ファイルがある場合にはロックファイル作成が失敗するようにして作成するには例えば次の方法がある。

- `mkdir` ロックファイル
- `ln -s` ダミーファイル ロックファイル
- `ln` ダミーファイル ロックファイル
- `(set -C; : >ロックファイル)`

ポイントは、アトミックに（単一操作で）作るという点である。つまり存在確認処理と作成処理が同時ということだ。もしこれから作りたい名前のファイルが存在しないことを確認できて、いざ作成しようとした時に他のプロセスに素早く作成されてしまったら、ロックファイルを上書きできてしまうのでアクセス権が唯一のものではなくなってしまう。

排他ロックコマンド `pexlock` では、今列挙した 4 つのうち最後の方法を用いている。後で紹介する共有ロックコマンドでは複数のアクセス権を管理するためにディレクトリー (`mkdir`) を用いており、それと区別させるためだ。

その 2. 共有ロック・セマフォはどうやるか

共有ロックとは、そのロックを申請した全てのプロセスでアクセス権を共有するためのロックだ。自分がファイルを読み込んでいる間、他のプロセスもそれを読み込むだけなら許すが、書き込み許さない、というプロセス同士がアクセス権を共有したい場合などに用いる。

一方セマフォとは、共有ロックの最大共有数を制限するロックだ。物理デバイスの数だけプロセスを同時に走らせたい場合などに用いる。セマフォは共有ロックの応用で実現できるため、ここでまとめて説明する。

排他ロックファイルに比べるとだいぶ複雑であるが、まず共有ロックファイル（ディレクトリー）には次の構造を持たせることとする。

■pshlock が扱う共有ロックファイル（HOGE という名前とする）の構造

```

LOCKDIR/           ← ロックファイル管理ディレクトリー
|
+-HOGE/            ← ロックファイルー式（親ディレクトリー）
|
+-HOGE/            ← ロックファイルー式（同名の子ディレクトリー）
|                  ・このディレクトリーのハードリンク数-2 が共有数
|
+-uniq_num1.pid1/ ← 共有ロックをかける度に作成するサブロック名
+-uniq_num2.pid2/  （一意な番号 + 呼出元プロセス ID）
:
|
+ modifying ← 上記のサブロック名ディレクトリーを追加削除する際の
               アクセス権のための排他ロックファイル（作業時のみ存在）

```

これにはいくつか工夫が凝らしてある。

■同名で二重化されたディレクトリー なぜ同名のディレクトリーを二重に作っているのか。これは共有ロックファイル（ディレクトリー）をアトミックに作るための巧妙な仕掛けである。

後で改めて説明するが、共有ロックディレクトリーの中には、共有中のプロセスによって一意に作られたディレクトリー（サブロックディレクトリー）が必ず存在しなければならない。現在共有中のプロセス数を示すためである。それゆえ、もし何も考えず本番のロック管理ディレクトリーに直接に新規作成してしまうと、作成した瞬間の共有ロックディレクトリーは空であるため、共有プロセス数が 0（もはや誰も共有していない）と見なされて削除される恐れがある。よって本番のロック管理ディレクトリーに直接作るとは避けなければならない。

そこで、予め別の安全な場所でサブロックディレクトリーまで中身を作っておき、mv コマンドを用いて本番ディレクトリーに移動させる。ところがもし移動先に既存の共有ロックディレクトリーがあると通常 mv コマンドは、その共有ロックディレクトリーのサブディレクトリーとして移動を成功させてしまう。共有ロックディレクトリーの直下にわざわざ同名のディレクトリーを置くのはこの問題への対策である。同名のディレクトリーが直下にあれば、mv コマンドも移動を諦めてくれる。

■中に < 固有番号 + ロック要求プロセス ID > のディレクトリーを作る これもまた巧妙な仕掛けだ。先程説明した二階層同名ロックディレクトリーの下層側（子）に、共有ロックを希望する各プロセスがさらに 1 つディレクトリーを作る。この際、ディレクトリー名が衝突しないように < 固有番号 + ロック要求プロセス ID > という命名規則（固有番号は、作成日時と pshlock コマンドプロセス ID に基づいて作ればよい）による一意な名称（サブロック名）を付ける。

目的は先程も述べたが、現在の共有プロセス数を把握するためである。共有数が把握できれば後述するセマフォ（共有数に制限を設ける）を実現できるし、また共有数 0 になった際に共有ロックディレクトリー自体を削除するという判断もできる。

では具体的にどうやって共有数を把握するか？ 中に作成したディレクトリー数を素直に数えるという方法もあるが、もっと軽い方法がある。共有ロックディレクトリー（子）のハードリンク数を見るという方法だ。

`ls -ld 共有ロックディレクトリー（子）`

を実行した時、2 列目に表示される数字がそれである。この数字から 2 を引くと、直下のサブディレクトリーの数になる^{*17}。従って上記のコマンドで 2 列目の数を取得すれば、いちいち全部数えずとも、共有数を計算でき

^{*17} ディレクトリーを作成すると、必ず自分自身を示す“.”と、親ディレクトリーを示す“..”が作成される。これらこそが数字を 2 つ

るのである。

■共有数を増減・参照する際は、更に排他ロック

- 共有数が0だったら共有ロックディレクトリーを削除する。
- 共有数が上限に達したらそれ以上の共有を拒否する（セマフォ制御）。

といった操作は、どうしてもアトミックに行うことができない。共有数を調べて処理を決めようとしている時に共有数が変化してしまう恐れがあるからだ。これを防ぐため、共有数を参照する時と共有数を増減させる時は、そこで更に排他ロックを掛けなければならない。

共有数を参照したい場合は今列挙したとおりだが、増減させたい時というのは次の場合である。

- 共有ロックを追加したい場合
- 共有ロックを削除したい場合
- 共有ロックディレクトリー（子）内のサブロック名ディレクトリーのうち古いものを、基本ルール2に従って一斉削除したい場合

以上、列挙した操作を行う場合に作る排他ロックファイルが

共有ロックディレクトリー（子）/modifying

というファイルである。

共有ロック・セマフォの実装のまとめ

以上の理屈に基づいてコードに起こしてみたものをいくつか例示する。実際のコードには異常系の処理等があり、これより込み入っているが、わかりやすくするためにそれらは書いていない。

■共有ロックファイル新規作成 まずは共有ロックファイル新規作成のコードを見てみよう。安全な場所で共有ロックディレクトリーを作成し、本番ディレクトリーに mv コマンドで移動している。もし mv に失敗した場合は、共有ロックディレクトリーが既に存在していることを意味しているので、次に例示するシェル関数 `add_shlock()` へ進む。

大きく見せている原因であって、今作ったディレクトリーに対するハードリンクなのである。

```

LOCKDIR="/PATH/TO/LOCKDIR" # ロックの管理を行うディレクトリー
lockname="ロック名"        # 共有ロック名
MAX_SHARING_PROCS=上限数   # セマフォモードの場合に使う上限数

# 安全な場所で共有ロックファイルを新規作成
callerpid=$(ps -o pid,ppid | awk ' $2=="$${print $1;exit}')
sublockname=$(date +%Y%m%d%H%M%S).$$. $callerpid
tmpdir="$LOCKDIR/.preshlock.$sublockname"
shlockdir_pre="$tmpdir/$lockname/$lockname/$sublockname"
mkdir -p $shlockdir_pre

# 本番ディレクトリーへの移動を試みる
try=3 # リトライ数
while [ $try -gt 0 ]; do
    # mv に成功したら新規作成成功
    mv $shlockdir_pre $LOCKDIR 2>/dev/null && {
        echo "$LOCKDIR/$lockname/$lockname/$sublockname" # 後で削除できるよう、
        break                                             # サブロック名ファイルパスを返す
    }

    # 失敗したら追加作成モードで試みる
    add_shlock && break # add_shlock() の中身は別途説明

    try=$((try-1))
    [ $try -gt 0 ] && sleep 1                                # リトライする場合は 1 秒待つ
done
case $try in 0) echo "timeout, try again later";exit 1;; esac

# 安全な場所として作ったディレクトリーを削除
rm -rf "$tmpdir"

```

尚、「安全な場所」を確保するためにロックファイルディレクトリーの中に “.preshlock. サブロック名” という一意なディレクトリーを作っている。そのため、`^¥.preshlock¥.[0-9.]+$` という正規表現に該当するロック名は予約名として使用禁止とする。

■共有ロックファイル（ディレクトリー）追加作成 前記のコードで共有ロックディレクトリーの新規作成に失敗した場合には次に記すシェル関数 `add_shlock()` が呼び出される。

共有数を増減・参照するのでまず排他ロックファイル `modifying` を作り、共有数が上限に達していないか調べ、達していなければサブロック名ディレクトリーを 1 つ作る。最後に排他ロックファイルを消すのも忘れぬようにする。

```

add_shlock() {
    # 共有数アクセス権取得（失敗したらシエル関数終了）
    (set -C; : >${LOCKDIR}/${lockname}/${lockname}/modifying) || return 1

    # 共有数 (=共有ロックファイル (子) の数-2) が制限を超えていないか
    # ※ この処理はセマフォ制御の場合のみ
    n=$(ls -dl ${LOCKDIR}/${lockname}/${lockname} | awk '{ $2-2 }')
    [ $n -ge $MAX_SHARING_PROCS ] || return 1 # 超過時は関数終了

    # 共有ロック追加
    sublockname=$(date +%Y%m%d%H%M%S).$$. $callerpid
    mkdir $tmpdir/${lockname}/${lockname}/${sublockname}
    echo "${LOCKDIR}/${lockname}/${lockname}/${sublockname}" # 後で削除できるよう、
                                                                # サブロック名パスを返す

    # 共有数アクセス権解放
    rm ${LOCKDIR}/${lockname}/${lockname}/modifying
}

```

■共有ロックファイル（ディレクトリー）削除 自分で作った共有ロックを削除する場合は、ロック成功時に渡されたロックファイル（サブロック名まで含む）のフルパスを渡す。それに基づいてサブロック名ディレクトリーを削除、更に古いディレクトリーも削除した結果、共有数 0 になっていたら共有ロックディレクトリーごと削除する。

```

lockfile="ここにロックファイル名（サブロック名まで含む）"

try=3 # リトライ数
while [ $try -gt 0 ]; do
    # 共有数アクセス権取得
    (set -C; : >${lockfile#*/}/modifying) && break

    try=$((try-1))
    [ $try -gt 0 ] && sleep 1
done
case $try in 0) echo "timeout, try again later"; exit 1;; esac

# ロック解除対象のサブロック名ディレクトリーを削除
rmdir $lockfile

# 共有数 (=共有ロックファイル (子) の数-2) が 0 なら共有ロック dir 自身を削除
n=$(ls -dl ${lockfile#*/} | awk '{ $2-2 }')
if [ $n -le 0 ]; then
    rm -rf ${lockfile#*/}/..
else
    # 共有数が 0 でなければ、共有数アクセス権を解放するのみ
    rm ${lockfile#*/}/modifying
fi

```

参照

→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 3.12 デバッグってどうやってるの？

問題

シェルスクリプトの IDE（統合開発環境）なんて聞いたこと無い（テキストエディターまでは聞いたことがあるが）。

シェルスクリプトのプログラムに対しては、一体どうやってデバッグを行うのか？

回答

まず基本的な心構えとして、プログラムを新規に書いている段階では、一ステップ書いては動作が正しいか確認するべきだろう。もちろんリリース後に不具合が見つかった時にもデバッグはするのだが、リリース前に一気にデバッグはしない。結果的にその方が制作が早いと感じているからだ。

では具体的にはどんな手段でデバッグをするかといえば、

- 連携しているアプリケーションの各種ログを確認する。
- 気になる箇所にテストコードを仕込んで検証する。
- パケットアナライザーでやりとりされるデータを確認する（CGI の場合）。
- パイプで繋がれているコマンド間に `tee` コマンドを仕込み、そこを流れるデータの途中経過をファイルに書き出して検証する。
- シェルスクリプトの実行ログ（`set -vx`）を取るようにして動作を検証する。

など、ごく普通のものである。ステップ実行をする手段がないくらいであとは IDE と比べても特に遜色は無いと思う。

ごく普通といいながらも、最後の 2 つはシェルスクリプト独特の手段なので解説しておこう。

tee コマンド仕込みデバッグ

これは、パイプ “|” で繋がれたいくつものコマンドでデータを加工していくようなプログラムのデバッグに役立つテクニックだ。

例えば数字マジックで、「次の計算を行うと答えが必ず 1089 になるよ」と言われ、

1. 好きな 3 桁の数 a （ただし数字は 3 桁とも違うものにする）を思い浮かべる。
2. その 3 桁の数 a の左右を入れ替えた数 b を作る。
3. a と b の差 x を求める（2 桁になったら 0 埋めして 3 桁にする）。
4. その 3 桁の数 x の左右を入れ替えた数 y を作る。
5. x と y の和を求めると……

本当かどうか確かめるために次のようなプログラムを組んだとしよう。

■数字のマジックを検証するプログラム magic.sh

```
#!/bin/sh

# 1) 好きな3桁の数 a を引数から受け取る                                #
echo $1                                                                |
# 2) その3桁の数 a の左右を入れ替えた数 b を作る                        #
awk '{print $1, substr($1,3,1) substr($1,2,1) substr($1,1,1)}' |
# 3) a と b の差 x を求める (3桁に0パディング)                          #
awk '{printf("%03d\n", $1-$2)}' |
# 4) その3桁の数 x の左右を入れ替えた数 y を作る                        #
awk '{print $1, substr($1,3,1) substr($1,2,1) substr($1,1,1)}' |
# 5) x と y の和を求めると……、必ず 1089 になるはず                    #
awk '{print $1+$2}'
```

ところが、試しに 123 という数字を与えたところ、-107 という結果が出てきた。

```
$ ./magic.sh 123 ↵
-107
$
```

何か間違っている！というわけでデバッグをすることにした。

「パイプを流れるデータを追いかけてみよう」ということで、tee コマンドを各行に挟んだデバッグプログラムを生成し、

■数字のマジックを検証するプログラム magic.sh のデバッグ版

```
#!/bin/sh

# 1) 好きな3桁の数 a を引数から受け取る                                #
echo $1                                                                | tee step1 |
# 2) その3桁の数 a の左右を入れ替えた数 b を作る                        #
awk '{print $1, substr($1,3,1) substr($1,2,1) substr($1,1,1)}' | tee step2 |
# 3) a と b の差 x を求める (3桁に0パディング)                          #
awk '{printf("%03d\n", $1-$2)}' | tee step3 |
# 4) その3桁の数 x の左右を入れ替えた数 y を作る                        #
awk '{print $1, substr($1,3,1) substr($1,2,1) substr($1,1,1)}' | tee step4 |
# 5) x と y の和を求めると……、必ず 1089 になるはず                    #
awk '{print $1+$2}'
```

このプログラムを実行後、step4 を見てみたら原因が判明した。

```
$ cat step4 ↵
-198 91-
$
```

a と b の差を求めた際に負の値になったらそれを除去し忘れていたことだった。そこで $a - b$ の計算後に負号を取り除くコードを追加したところ。

■数字のマジックを検証するプログラム magic.sh のデバッグ版

```
#!/bin/sh

# 1) 好きな 3 桁の数 a を引数から受け取る                #
echo $1                                                    |
# 2) その 3 桁の数 a の左右を入れ替えた数 b を作る        #
awk '{print $1, substr($1,3,1) substr($1,2,1) substr($1,1,1)}' |
# 3) a と b の差 x を求める (3 桁に 0 パディング)          #
awk '{printf("%03d¥n", $1-$2)}'                            |
tr -d ' '                                                  | # 追加
# 4) その 3 桁の数 x の左右を入れ替えた数 y を作る        #
awk '{print $1, substr($1,3,1) substr($1,2,1) substr($1,1,1)}' |
# 5) x と y の和を求めると……、必ず 1089 になるはず      #
awk '{print $1+$2}'
```

正しく動くようになった。

```
$ ./magic.sh 123
1089
$
```

これが tee コマンド仕込みデバッグである。

実行ログ収集デバッグ

tee コマンドがパイプを流れるデータに注目するデバッグ方法であるのに対し、こちらはシェルスクリプトの動作に注目するデバッグ方法である。具体的にはシェル変数や制御構文 (if, for, while など) が正しく動作しているを診るのに役立つ。

先程の数字のマジックをシェル変数ベースで書いたプログラム (同様のバグが残っている) があり、デバッグすることになった。このシェルスクリプトではパイプを一切使っていないので tee コマンド仕込みデバッグが通しない。そこで、冒頭の 2 行に exec と set のおまじないを書いて実行する。

■数字のマジックを検証するプログラム magic2.sh

```
#!/bin/sh

# 実行ログを取得
exec 2>/PATH/T0/logfile.$$ .txt # 標準エラー出力の内容をファイルに書き出す
set -vxx                         # 実行ログの標準エラー出力への書き出しを開始する
# (この行以降のシェルスクリプトの動作が/PATH/T0/logfile.$$ .txt に記録される)

a=$1
b=$(awk "BEGIN{print substr($a,3,1) substr($a,2,1) substr($a,1,1)}")
x=$(printf '%03d' $((a-b)))
y=$(awk "BEGIN{print substr($x,3,1) substr($x,2,1) substr($x,1,1)}")
echo $((x+y))
```

実行後に実行ログの中身を見るとこのようになっていた。

■シェルスクリプト magic2.sh の実行ログ

```
# (この行以降のシェルスクリプトの動作が/PATH/TO/logfile.txt に記録される)

a=$1
+ a=123
b=$(awk "BEGIN{print substr($a,3,1) substr($a,2,1) substr($a,1,1)}")
awk "BEGIN{print substr($a,3,1) substr($a,2,1) substr($a,1,1)}"
awk "BEGIN{print substr($a,3,1) substr($a,2,1) substr($a,1,1)}"
++ awk 'BEGIN{print substr(123,3,1) substr(123,2,1) substr(123,1,1)}'
+ b=321
x=$(printf '%03d' $((a-b)))
printf '%03d' $((a-b))
printf '%03d' $((a-b))
++ printf '%03d -198
+ x=-198
y=$(awk "BEGIN{print substr($x,3,1) substr($x,2,1) substr($x,1,1)}")
awk "BEGIN{print substr($x,3,1) substr($x,2,1) substr($x,1,1)}"
awk "BEGIN{print substr($x,3,1) substr($x,2,1) substr($x,1,1)}"
++ awk 'BEGIN{print substr(-198,3,1) substr(-198,2,1) substr(-198,1,1)}'
+ y=91-
echo $((x+y))
magic2.sh: line 12: 91-: syntax error: operand expected (error token is "-")
```

先頭に“+”がない行は、その時点で読み込んだコードやその他標準エラー出力への出力文字列である。一方“+”で始まる行はコードの実行結果である。シェル変数も展開された状態で出力され、どの時点でどのように実行されたが一目でわかる。

この実行ログの最後の行を見ると、magic2.sh の 12 行目で実行時エラーが発生しているのがわかるので、これを手掛かりにデバッグしていくのである。

解説

tee コマンドについて

tee コマンドの由来は、T 字型のパイプである。T 字型のパイプは、入ってきた流体が 2 方向に分岐する。tee コマンドはこれと同様に、標準入力から入ってきたデータを一方はそのまま標準出力に送り、もう一方は引数で指定されたファイルに送る。(ただし本物の T 字型パイプと違ってデータ量は半減しない)

これは UNIX において、パイプという仕組みと並んで偉大な発明品だと思う。

set コマンドの-v と-x オプションについて

シェルの内部コマンド set におけるこれら 2 つのオプションは、まさにデバッグのために存在するといっても過言ではないのではなかろうか。

先程示した実際の実行ログを見ればわかるように、“-v” オプションは読み込んだシェルスクリプトのコードをそのまま標準エラー出力へ送るもの、“-x” オプションは実行したコード文字列を（シェル変数を展開しながら）標準エラー出力へ送るものである。シェルにはステップ実行をしてくれる機能は無いが、実行経過を各行ログに書き出してくれるのは様々な言語を見渡しても珍しい機能ではないだろうか。

ちなみにシェルスクリプトの途中で

```
set +vx
```

と書けば、その行以降は実行ログに出力されないし、併せて

```
exec 2>/dev/stderr
```

と書けば本来の標準エラー出力に戻る^{*18}。

実行ログを恒久的に残すか

実行ログはデバッグのために紹介したが、私はデバッグ作業の時のみならず、リリース後も日頃から実行ログを取り、恒久的に残すようにしている。だが、その是非を巡っては2つの意見が対立する。

■恒久的に残す利点 最大の利点は、緊急事態に陥った時でも、素早く平常を取り戻すのに役立つ点にある。

例えば、作ったプログラムに100万回に1度の割合で起こるバグが潜んでいたとしよう。しかし一度起こると大損害をもたらす最悪のバグだとする。

ある日そのバグが発生してしまったために、血眼になってデバッグする羽目になった。ところが、実行ログをとっていなかったので何も手がかりがない！再現するにも100万回に一度の現象などそう簡単には起こらない。一体、このデバッグ地獄は作業はいつ終わるんだろうか……。恒久的に実行ログを残しておけばこうした地獄を見ずに済む。実行ログも「ログ」の一種であり、ログとは本来そういう目的のものである。

現実世界というのは、いくら自分が正しいプログラムを書いても、外部システムやハードウェア、あるいは人為的な問題により想定外の事故が起こる。実行ログの恒久的保存という発想も、こういう事態に散々泣かされてきたからこそ、生まれたものである。

■恒久的に残す危険性 実行ログ保存の恩恵の裏には2つの脅威がある。

1つはログファイルによるディスク空き領域の圧迫である。1回のWebアクセスで1行～数行程度のWebアクセスログの成長に比べると実行ログの成長速度は何百倍にもなる。従って、異常な頻度でアクセスを受ければディスク空き領域が食い尽くされてサービス不能に陥る恐れがあるということだ。

もう1つは機密情報の漏洩である。実行ログには動作中のあらゆる情報が残る。先程示したように、シェル変数の内容は丸見えだ。例えばログイン認証のCGIスクリプトであれば、クライアントから送られてきたID、パスワードが平文で残ってしまう。無関係な人にそのログファイルを開かれてしまったらあなたのプログラマーとしての信用は地に墜ちるだろう。

■あるセキュリティー専門家のアドバイス ディスク空き領域の圧迫に対する対策は容易だ。残量が一定以下であることを検出したらログ収集を中止したり、過去のものから削除するようにプログラムを組めばよい。より深い問題は、情報漏洩対策だ。

この問題に対し、私はセキュリティー専門家から

ログファイルにgzip等で圧縮を掛けて保存するだけでも効果があります。

とアドバイスを受けた。

悪意ある人間は知能をもっており、そういう相手からログファイルを守るのは確かに困難だ。しかし、手当たり次第に情報を暴露しようとするマルウェアに侵される可能性の方がよほど高く、それらには知能が低く、大抵バイナリーファイルはバイナリーのまま暴露するため、途中で壊れる可能性が高いからだという。もちろん、将来マルウェアの知能が上がったら暗号化の検討も必要であろうが、確かに現状での妥協点だと思った。

^{*18} ファイルディスクリプターを熟知しているのであれば、冒頭で `exec 3>&2 2>logfile` などと書いて本来の標準エラー出力を別のファイルディスクリプターに退避させておき、最後に `exec 2>&3 3&-` とやるのが望ましい。

そこで私は、zpipe というコマンドを作った*¹⁹。これはコマンド呼び出し元のプロセスが生存している間だけ存在するような、名前付きパイプを作る（=mkfifo する）コマンドで、その名前付きパイプの先には圧縮コマンド（compress）が待ち構え、随時圧縮しながらファイルに書き落とすという仕組みだ。

使い方はつぎのとおり。

■zpipe コマンドを使った圧縮実行ログ取得方法

```
#!/bin/sh

# 実行ログを取得
# zpipe: 第1引数=作りたい名前付きパイプのパス
#        第2引数=最終的に書き落としてほしいファイルのパス
zpipe /PATH/T0/named_pipe /PATH/T0/logfile.$$ .txt.Z
[ $? -eq 0 ] || exit 1          # zpipe に失敗したら中断
exec 2>/PATH/T0/named_pipe     # 今作った名前付きパイプに書き込ませる
set -vx                        # 走行ログ取得開始

# 以降の内容が実行ログに記録される
:
:
(zpipe を実行したプログラムが終了したら、named_pipe は自動消滅する)
```

もちろん平文の実行ログを作った後で、cron でまとめて圧縮するシェルスクリプトを作ってもよいのだが。このコマンドを使えば平文での生存期間をゼロにすることが可能である。

*¹⁹ <https://github.com/ShellShoccar-jpn/misc-tools/blob/master/zpipe>

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「メトロバイパー」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
    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 "%n";
    }
}'

```

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。

このコードをいちいち書くのも面倒であると思うので、コマンド化したものを GitHub で公開^{*1}した。そちらを使ってもいい。

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^{*2}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。このツールは x86(32bit) 向けのアセンブリで書かれており、なんと 64bit 環境非対応なのだ。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

^{*1} <https://github.com/ShellShoccar-jpn/misc-tools/blob/master/urldecode>

^{*2} <http://www.whizkidtech.redprince.net/urlendec/>

る。(かつての筆者)

参照

→ RFC 3986 文書^{*3}

→ 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

Web API 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か頭に “%” 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
    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 "¥n";
    }
}
```

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。

こちらのコードもいちいち書くのも面倒だと思うので、コマンド化したものを GitHub で公開^{*4}した。そ

^{*3} <http://www.ietf.org/rfc/rfc3986.txt>

^{*4} <https://github.com/ShellShoccar-jpn/misc-tools/blob/master/urlencode>

ちらを使ってもいい。

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 原理主義者ならと一ぜん禁じ手だ。

参照

→レシピ 4.1 (URL デコードする)

→RFC 3986 文書^{*5}

レシピ 4.3 Base64 エンコード・デコードする

問題

メールを扱うシェルスクリプトを書きたい。しかしメールでは、Base64 エンコードやデコードをしなければならない場面が多数ある。どうやればいいか?

回答

Linux あたりだと base64 というコマンドが標準でインストールされていたりするのでそれを使う手もあるが、残念ながら POSIX 準拠コマンドではない。とはいえ Base64 の変換アルゴリズムはさほど複雑ではなく、公開されている仕様 (RFC 3548) や既存の base64 コマンドに準拠するものを POSIX の範囲で実装したがあるのでそれを使う。

Base64 コマンドの使い方

どのようにして実装したのかは解説の項で説明するとして、まずは使ってみよう。

^{*5} <http://www.ietf.org/rfc/rfc3986.txt>

Base64 コマンドを POSIX の範囲で書き直し、GitHub に公開したので、手元の環境になれば下記のページからそれをダウンロードする。

```
https://github.com/ShellShoccar-jpn/misc-tools/blob/master/base64
```

ダウンロードして実行権限を与えたら、使ってみる。標準入出力経由でデータを渡すと Base64 に変換される。

```
$ echo 'ShellShoccar' | ./base64 ↵
U2h1bGxTaG9jY2FyCg==
$
```

-d オプションを付ければデコードになるので、先程 Base64 変換した文字列を渡せば元の文字列が出てくる。

```
$ echo 'U2h1bGxTaG9jY2FyCg==' | ./base64 -d ↵
ShellShoccar
$
```

エンコードとデコードは互いに逆変換なので、それらを連続して通せば元の文字列が出力される。

```
$ echo 'ShellShoccar' | ./base64 | ./base64 -d ↵
ShellShoccar
$
```

解説

Base64 の仕様は複雑ではないので素直に実装すればよい、とはいえ、シェルスクリプトにとっては大きな問題がある。

それは Base64 が対象としているデータはテキストデータのみならずバイナリーデータもあるのだが、シェルスクリプトでは NULL 文字 < 0x00 > を扱うのが大変なのである。しかしこの問題は、レシピ 3.10 (バイナリーデータを扱う) で克服した。先に紹介した POSIX 版 base64 コマンドは、そのレシピで示したコードに Base64 の仕様を追加して作られている。

参照

→レシピ 3.10 (バイナリーデータを扱う)

→レシピ 4.8 (シェルスクリプトでメール送信)

レシピ 4.4 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存在するので、まずこれらをダウンロード^{*6}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。

■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
```

なお、元のデータに漢字や記号が含まれていて、それが URL エンコードされたものでももちろん構わない。そういった文字列は cgi-name コマンドがデコードについても済ませてくれる。

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&...
```

^{*6} <https://github.com/ShellShoccar-jpn/Open-usp-Tukubai/blob/master/COMMANDS.SH/cgi-name>、及び

<https://github.com/ShellShoccar-jpn/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。

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、そのファイルから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^{*7}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 4.5 (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 4.5 (CGI 変数の取得 (POST メソッド編))

レシピ 4.5 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。

^{*7} https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_cgi-name

回答

基本的にはレシピ 4.4（CGI 変数の取得 (GET メソッド編)）と同じである。よってそちらで出てきた 2 つのコマンド（“cgi-name” 及び “namread”）をダウンロード^{*8}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。

■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 4.4（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

^{*8} <https://github.com/ShellShoccar-jpn/Open-usp-Tukubai/blob/master/COMMANDS.SH/cgi-name>、及び
<https://github.com/ShellShoccar-jpn/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。

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。そのためこのようなやり方を推奨している。

参照

→レシピ 4.4（CGI 変数の取得 (GET メソッド編)）

→レシピ 4.6（Web ブラウザーからのファイルアップロード）

レシピ 4.6 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、受け取るにはどうすればいいか？

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存在する。まずこれをダウンロード^{*9}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、この形式のデータを取得するためのシェルスクリプトは次のようになる。

^{*9} <https://github.com/ShellShoccar-jpn/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 -Ei '^[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になるため、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 4.5 (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、データの中身とは区別できるようにしている。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*10ので、バイナリーデータの取り出し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 4.4 (CGI 変数の取得 (GET メソッド編))

→レシピ 4.5 (CGI 変数の取得 (POST メソッド編))

→レシピ 1.7 (改行無し終端テキストを扱う)

レシピ 4.7 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 原理主義者の 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。

*10 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}
    </style>
    <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<div id="clock">～</div>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、Web ブラウザー上で Ajax 処理を行う JavaScript(CLOCK.JS) 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
// (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 = 'http://somewhere/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。

ほぼ同じ内容のファイルを GitHub 上に公開^{*11}してあるので、この 3 つのファイルを適宜 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.HTM 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。

^{*11} https://github.com/ShellShoccar-jpn/Ajax_demo

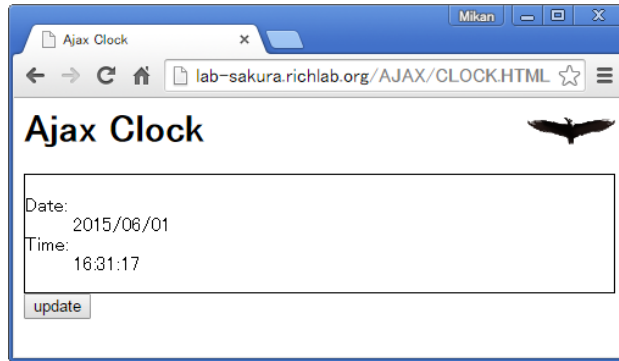


図 4.1 Ajax デモプログラムの動作画面

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 初心者はずそ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 の最後尾に UNIX 時間 (Date オブジェクトでそれをミリ秒単位に求める) を値にとるダミー CGI 変数を置くことで、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 スクリプトは時刻データを XML や JSON で表現したもので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、苦勞して 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。その分サーバー側のプログラミングが大変になると思うかもしれないが、レシピ 4.10 (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、もっと複雑な HTML でも簡単に生成できるのだ。

ただし困ったことに **IE8** は、`<select>` タグには innerHTML プロパティ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^{*12}

→ レシピ 4.10 (HTML テーブルを簡単綺麗に生成する)

レシピ 4.8 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:ヘッダー付きで送ることも日本語メールを送ることもできない^{*13}。

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) — 必要な非 POSIX コマンド

表 4.2 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。

もし、Subject: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、メール本文を POSIX 標準の `iconv` コマンドで JIS に変換して済ませるという手もあるが、ここでは割愛する。

^{*12} <https://developer.mozilla.org/ja/docs/Web/API/XMLHttpRequest>

^{*13} 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。

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。
base64	日本語文字エンコード（本文や Subject:欄等）のため	パフォーマンスでは及ばないが、POSIX の範囲で書いた base64 コマンドあり→レシピ 4.3

Step(1/3) – 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^{*14}。

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。

メールテキストができれば送信する。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。

^{*14} 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§=0> である。

step(2/3) — 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（ただし UTF-8）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="UTF-8"
Content-Transfer-Encoding: base64
```

やあ、これ読める？

本文に日本語文字が入った他に、ヘッダーセクションに `Content-Type: text/plain;charset="UTF-8"` と `Content-Transfer-Encoding: base64` を追加した。これは、本文が UTF-8 エンコードされていること、さらにそれが Base64 エンコードされているということを知らせるためだ。つまり、本文はこれから Base64 エンコードをするのだ。その理由は、UTF-8 は <0x80> 以上の文字を含んでおり、そのままではメールサーバーを通過できないためだ。

具体的には、次のようなコマンドを打てばよい。

```
$ cat mail2.txt | while read -r line; do
>   case "$line" in
>     '') echo
>         cat | base64
>         ;;
>     *) printf '%s¥n' "$line"
>         ;;
>   esac
> done
$
```

これは while ループでヘッダーセクション（最初の改行が現れるまで）をそのまま通し、本文セクションを base64 コマンドでエンコードする*15 というものだ。

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) — 件名や宛先も日本語化したメールを送る

From:や To:はまあ許せるとしても、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。

*15 while read ループ内で cat 等により標準入力を受け取ると、以降は read コマンドが標準入力データを受け取れなくなり、ループはその周で終わる。




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 UTF-8 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。(ただし文字エンコードはどちらも UTF-8 とする)

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

```
$ printf '%s\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
> base64 |  ← Base64 エンコードする
> xargs printf '=?UTF-8?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
=?UTF-8?B?440P440t440844CBZS1tYWlsIQ==?=$
```

■「あなた」をエンコード

```
$ printf '%s\n' 'あなた' | 
> base64 | 
> xargs printf '=?UTF-8?B?%s?=%n' 
=?UTF-8?B?44GC44Gq44Gf?=$
```

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 (printf '%s\n' を使うとよい)
2. Base64 エンコード
3. 生成された文字列の左端に “=?UTF-8?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: =?UTF-8?B?44GC44Gq44Gf?= <RECIEVER@example.com>
Subject: =?UTF-8?B?440P440t440844CBZS1tYWlsIQ==?=
Content-Type: text/plain; charset="UTF-8"
Content-Transfer-Encoding: base64
```

やあ、これ読める？

これを先程の mail2.txt と全く同じように送ればよい。今度は宛名も件名もきちんと日本語文字列で届いたことを確認してもらいたい。

step(3/3) の作業の完全自動化シェルスクリプトを公開

ここまでのチュートリアルを見て、やり方はわかったであろうがいちいち手作業で宛先や件名のエンコード文字列を貼る作業など面倒臭い。そこで、ヘッダー部分 (From:、Cc:、Reply-To:、Subject:) も本文も日本語で書かれたメールを与えるだけで必要なエンコードをしながら sendmail コマンドで送信してくれる **sendjpmail** コマンドを作り、GitHub に公開した。

<https://github.com/ShellShoccar-jpn/misc-tools/blob/master/sendjpmail>

sendmail コマンドを除けば、POSIX 準拠である。Base64 エンコーディングも自力で行っているので base64 コマンド無しでも動く (ホスト上にあればそちらを利用する)。

さて例えば次のように、ヘッダーや本文に日本語の含まれたメールファイル (Content-Type: や Content-Transfer-Encoding: ヘッダーも省略可) があった時、

■メールサンプル (mail4.txt)

```
From: わたし <SENDER@example.com>
To: あなた <RECIEVER@example.com>
Subject: ハロー、e-mail!
```

やあやあ、これも読める？

次のようにして (標準入力から渡すのも可) sendjpmail コマンドを一発叩けば、必要な Base64 エンコーディングや Content-Type: ヘッダー等付加を行って正しいメールを送ってくれる。

```
$ sendjpmail mail4.txt ↵
$
```

参照

→レシピ 4.3 (Base64 エンコード・デコードする)

→レシピ 3.8 (全角・半角文字の相互変換)

レシピ 4.9 メールマガジンを送る

問題

会員番号、性別、生年、姓、名、メールアドレスの記されたテキスト会員名簿（members.txt）がある。

```
0001 f 1927 町田 芹菜 serina@oda.odaq
0002 m 1941 海老名 歩 namihei@oda.odaq
0003 f 1929 大和 桜 sakura@enos.odaq
0004 f 1927 和泉 玉緒 tamao@oda.odaq
0005 m 1952 蛭田 蓮正 renscho@odawara.odaq
0006 m 1927 千歳 繁 shigeru@odawara.odaq
0007 f 1974 黒川 若葉 wakaba@tam.odaq
0008 ? ? 豪徳寺 タマ tama@oda.odaq
```

この人々にメールマガジンを送るにはどうすればよいか。ただし宛先には Bcc: フィールドを用い、会員同士では互いにアドレスが知られないようにしなければならない。

回答

先程のレシピ 4.8 で紹介した sendjpmail コマンドを活用すれば送れる。具体的な手順は次のとおりである。

0) 必要なコマンド

まず、次のコマンドを用意する。sendjpmail の中で sendmail コマンドを呼び出している以外は全て POSIX の範囲で書かれたシェルスクリプトである。下の 2 つのコマンドに関しては、使わなくても本件の問題をこなす

表 4.2 本件のメールマガジンを送るために用いるコマンド

コマンド名	目的	備考
sendjpmail	シェルスクリプトからメールを送るため	詳細および入手方法はレシピ 4.8 の Step(3/3) 参照
filehame	メールテンプレートにアドレス文字列をハメ込むため	Open usp Tukubai に存在する同名コマンドのシェルスクリプト版クローンである。
self	列の抽出を簡単に記述するため	Open usp Tukubai に存在する同名コマンドのシェルスクリプト版クローンである。

ことはできる。しかし、利用するとシェルスクリプトを簡潔に書くことができるため、使うことにした。それぞれのコマンドは下記の URL から入手可能だ。

<https://github.com/ShellShoccar-jpn/Open-usp-Tukubai/blob/master/COMMANDS.SH/filehame>
<https://github.com/ShellShoccar-jpn/Open-usp-Tukubai/blob/master/COMMANDS.SH/self>

1) メールのテンプレートを用意

一番肝心なメールマガジンの文面を作る。注意すべき点は次の3つ。

- ヘッダーセクションと本文セクションの間に必ず空行を挿むこと。
- マルチバイト文字を使う場合は UTF-8 にすること。
- 実際の宛名を書き込む BCC フィールドは、`###BCCFIELD###`というマクロ文字列にしておくこと。

以上の点に気をつけながら例えば次のようなテンプレートを作る。

■送信メールテンプレート (mailmag201507.txt)

```
From: 山谷 <sanya@staff.odaq>
To: <no-reply@staff.odaq>
###BCCFIELD###
Subject: 沿線日より 2015 年 7 月号
```

みなさんこんにちは、スタッフの山谷でございます。
今月は沿線で開催される花火大会情報を特集します。

■■■ 花火大会情報 ■■■

```
7/11 ○○花火大会
7/12 △△花火まつり
:
:
```

2) 一斉送信シェルスクリプトを作成

テンプレートが書けたら、続いて送信用のシェルスクリプトを作成する。

■メール一斉送信シェルスクリプト (sendmailmag.sh)

```
#!/bin/sh

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

cat members.txt          |
self 5                   | # メアド列だけ取り出す (awk '{print $5}' などと書き換え可)
sed 's/^.*$/ <&/'         | # メアド文字列をインデントしつつ"<>"で囲む
sed '1s/^/Bcc:/'         | # 最初の行にだけ、行頭に"Bcc:"を付ける
sed '$!s$/$/,'          | # 最後の行以外にだけ、行末にカンマを付ける
filehame -lBCCFIELD "$1" | # テンプレートにメアドをハメる
sendjpmail                # UTF-8 を 7bit 化してメールを一括送信
```

3) 送信

あとは、filemahe コマンドと、sendjpmail コマンドに実行権限とパスを通した状態で、

```
$ ./send_mailmag.sh mailmag201507.txt ↵
```

解説

この問題はレシピ 4.8 で紹介した sendjppmail コマンドの応用例である。

sendjppmail コマンド、もといそこから呼び出される sendmail コマンドは、メールテンプレートファイルに宛先をまとめて書き込んでおけば一回実行するだけでその全員にメールを送る。そこで、sed コマンドや Open usp Tukubai コマンドを駆使し、会員名簿ファイルから対象者全員分の Bcc: フィールドを生成し、テンプレートにハメ込んで送信しているわけである。

回答で示したシェルスクリプトには一つ注目すべき特徴がある。それは、このコードの中ではテンポラリーファイルもループ構文も一切使っていないという点だ。確かに呼び出し先のコマンド内にはループ処理が存在するが、そこに隠蔽したおかげで、このシェルスクリプト自体は非常に可読性の高いものになった。上から下へ読むだけで済むので可読性が非常に高いし、コメントだけ残せばそれが処理の手順書になってしまう。敢えて Open usp Tukubai の filehame や self コマンドを併用した理由は、それらを積極的に活用すれば可読性の高いコードが書けるということを示したかったからだ。

性別と年齢で宛先を絞り込む

元の会員情報テキストには、性別と年代の情報もあるのでこれで絞り込み、成人男性のみをターゲットにしたメールマガジンを送信する、といった応用も可能だ。

■成人男性限定配信用に改造してみる (sendmailmag2.sh)

```
#!/bin/sh

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

cat members.txt |
awk ' $3<="$(date +%Y)-20)" | # 成人 (20 歳以上) だけに絞り込む
    ' |
awk ' $2=="m" | # 男性だけに絞り込む
    ' |
self 5 |
sed 's/^.*$/ <&>/' |
sed '1s/^/Bcc:/' |
sed 's!s/$/,/' |
filehame -lBCCFIELD "$1" |
sendjppmail
```

最初の cat と AWK コマンド 2 つ、またその後ろの sed コマンド 3 つをそれぞれ 1 行にまとめろという声が聞こえてきそうだが、それはすべきではない。可読性が低下するし、最後にメール送信という桁違いにコストの高い処理が待ち構えているのに、ここでパフォーマンスに凌ぎを削る必要性を感じないからだ。

参照

→レシピ 4.3 (シェルスクリプトでメール送信)

→レシピ 3.8 (全角・半角文字の相互変換)

レシビ 4.10 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^{*16}し、次のチュートリアルをやってみてもらいたい。

mojihame コマンドチュートリアル

序章でも紹介した、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」^{*17}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

メトロパイパーでは 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」の駅名選択枝の表示だ。これらの選択枝は `<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^{*18}を用意する。

^{*16} <https://github.com/ShellShoccar-jpn/Open-usp-Tukubai/blob/master/COMMANDS.SH> にアクセスし、そこにある `mojihame`、`mojihame-h`、`mojihame-l`、`mojihame-p` の4つのソースコードをダウンロードする。

^{*17} <http://metropiper.com/>

^{*18} 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。

■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="-">--</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_SNUM_LIST”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_SNUM_LIST”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="-">--</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〉

```

左の列から順に、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^{*19}がこれだ。

^{*19} プログラム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。

■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/ー/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 コマンドまでの間で、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^{*20}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。

```
<!-- FROM_SELECT_BOX -->
<option value="-">-</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 4.7 (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。

この時押上駅には列車が 3 編成在線しており、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 (押上ー錦糸町)、錦糸町駅、駅間 (錦糸町ー住吉)、……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あ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。

^{*20} https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_mojihame

半蔵門線-住吉駅 の近接表示

2014年12月16日 18:51現在

今から 約73秒 前の情報です。



図 4.2 メトロパイパーの在線情報

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^{*21}は次のとおりである。

^{*21} 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。

■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”^{*22}の中の次の行で、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 mojihame コマンドの 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。

^{*22} https://github.com/ShellShoccar-jpn/metropiper/blob/master/CGI/GET_LOCINFO.AJAX.CGI 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^{*23}

→ レシピ 4.7 (Ajax で画面更新したい)

→ 「メトロパイパー」ソースコード^{*24}

レシピ 4.11 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 4.4（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 “;”

^{*23} https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_mojihame

^{*24} <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 で詳しく定義されているが、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^{*25}

→ レシピ 4.4 (CGI 変数の取得 (GET メソッド編))

→ レシピ 4.12 (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 4.12 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。しかしやるのがたくさんあるので、POSIX の範囲で実装した “`mkcookie`” コマンド^{*26}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。

^{*25} <http://tools.ietf.org/html/rfc6265>

^{*26} <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 (2015/06/01 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=example.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、セミコロン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);
}'
```

このように、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 4.11 (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^{*27}

→ RFC 2616 文書^{*28}

レシピ 4.13 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^{*29}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 分未満にアクセスすると、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 分未満にアクセスすると、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。

^{*27} <http://tools.ietf.org/html/rfc6265>

^{*28} <http://tools.ietf.org/html/rfc2616>

^{*29} mktemp コマンドは POSIX で規定されたコマンドではないのだが、POSIX の範囲で書き直したほぼ同等のコマンドを GitHub に公開した。→レシピ 5.24（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
Tmp=/tmp/tmp.$$                 # 一時ファイルの基本名
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-1}" |
                 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
    *)      mkdir -p $Dir_SESSION
            File_session=$(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)' ¥ (→次頁へ続く)

```


(→前頁からの続き)

```

$(date '+%Y %m %d %H %M %S') "$session_id"
> "$File_session"

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

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

    $msg
HTTP_RESPONSE

# --- 7) テンポラリーファイル削除 -----
rm -f $Tmp-*

```

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、レシピ 4.4（CGI 変数の取得 (GET メソッド編)）で紹介した `cgi-name` コマンドと `nameread` コマンド、及びレシピ 4.12（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。

なお、ほぼ同じ内容のプログラムを GitHub に公開^{*30}したので、そちらを使って試してみてもよい。

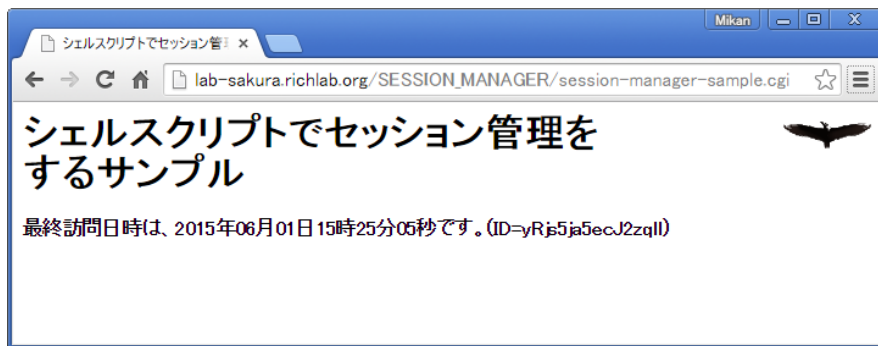


図 4.3 セッション管理デモプログラムの動作画面

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 HTTP リクエストヘッダー内にあれば、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。

^{*30} https://github.com/ShellShoccar-jpn/session_demo

- 4) セッションファイルの内容を見つ、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) セッションファイルを改めて 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^{*31}。

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=$Dir_SESSION/XXXXXXXXXXXXXXXXXXXXX ¥
                                lifemin=$SESSION_LIFETIME
                                )
case $? in 0) session_status='exist';; *) session_status='new';; esac
session_id=${File_session##*/}
```

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” にするだけでよい。

```
File_session=$(sessionf renew "$session_id" at=$Dir_SESSION/XXXXXXXXXXXXXXXXXXXXX ¥
                                lifemin=$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……

^{*31} <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。

```
$ 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 4.11 (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 4.12 (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 5.24 (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

他の章では冒頭で「問題」を提示し、それに対する「回答」という形で役立つレシピを様々紹介しているが、この章ではそういった具体的な問題に取り組むのではなく、どの UNIX 系 OS 上でも動くコードを記述するために都度確かめておきたい事項を辞典形式でまとめた。

次の 3 項目から構成される。

1. 文法・変数 シェル (Bourne Shell) 自身が解釈する構文や変数について、どの環境でも使える書き方を記している。
2. 正規表現 コマンドやその実装によって通用する正規表現のメタ文字の範囲が異なるため、それを体系的にまとめている。
3. コマンド POSIX の範囲で使えるコマンド、オプションはどの範囲かということをコマンドのアルファベット順にまとめている。

どの環境でも使えるシェルスクリプトを記述する際には本書を作業机に置いておき、確認したい項目をこまめに引くようにするとよいだろう。

尚、基本的には POSIX (“IEEE Std 1003.1”) に準拠させるという観点でまとめているが、一部例外もある。POSIX は様々な UNIX 系 OS が持つ仕様の共通部分を元に規定された経緯をもつが、OS 間で相反するために採用できなかった仕様というものが一部存在し、現存する OS にもこれを残しているものがある。従って POSIX に準拠していれば、UNIX 系 OS で 100% 通用するとは言えないため、そのような例外については個別に解説している。

また POSIX 文書には記されていない、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 環境変数等の初期化

シェルスクリプトが起動した時、現時点で設定されている環境変数等に影響されて意図しない動作をするようでは、「どの環境でも動く」という趣旨を満たしているとは言えない。従って環境変数等は一般的な内容に初期化しておくべきである。

次のコードをシェルスクリプトの冒頭に書いておくことをお勧めする。

```
set -u
umask 0022
PATH='/usr/bin:/bin'
IFS=$(printf ' %t%n_'); IFS=${IFS%_}
export IFS LC_ALL=C LANG=C PATH
```

最初の行の `set -u` は、環境変数等の初期化とは若干趣旨が異なるが、未定義の変数を読み出そうとした場合にエラー終了させるための宣言であり、strict なコードを記述したい場合に役立つだろう。

レシピ 5.2 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。同じことをがやりたいのならレシピ 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は、次のとおりである。

表 5.1 POSIX で対応しているシェル変数の展開書式

書式	条件と動作
<code>\${var:-word}</code>	変数 <code>\$var</code> が未定義か空文字の場合は、文字列 “word” が読み出される。
<code>\${var-word}</code>	変数 <code>\$var</code> が未定義の場合は、文字列 “word” が読み出される。
<code>\${var:=word}</code>	変数 <code>\$var</code> が未定義か空文字の場合は、文字列 “word” が読み出されると共に変数 <code>\$var</code> にも代入する。
<code>\${var=word}</code>	変数 <code>\$var</code> が未定義の場合は、文字列 “word” が読み出されると共に変数 <code>\$var</code> にも代入する。
<code>\${var:?word}</code>	変数 <code>\$var</code> が未定義か空文字の場合は、文字列 “word” が読み出されると共に、エラー扱い（戻り値 <code>\$?</code> が非ゼロ）にする。
<code>\${var?word}</code>	変数 <code>\$var</code> が未定義の場合は、文字列 “word” が読み出されると共に、エラー扱い（戻り値 <code>\$?</code> が非ゼロ）にする。
<code>\${var:+word}</code>	変数 <code>\$var</code> が未定義でも空文字でもなければ、文字列 “word” が読み出される。
<code>\${var+word}</code>	変数 <code>\$var</code> が未定義でなければ、文字列 “word” が読み出される。
<code>\${#var}</code>	変数 <code>\$var</code> の文字数（現在のロケールに応じた文字数）が読み出される。
<code>\${var#pattern}</code>	変数 <code>\$var</code> に格納されている文字列の左端に、パターン “pattern” があれば、それが最小マッチングで切り落とされて読み出される。例えば <code>var="/a/b/c"; echo "\${var#*/}"</code> の場合、“a/b/c” が読み出される。
<code>\${var##pattern}</code>	変数 <code>\$var</code> に格納されている文字列の左端に、パターン “pattern” があれば、それが最大マッチングで切り落とされて読み出される。例えば <code>var="/a/b/c"; echo "\${var###*/}"</code> の場合、“c” が読み出される。
<code>\${var%pattern}</code>	変数 <code>\$var</code> に格納されている文字列の右端に、パターン “pattern” があれば、それが、最小マッチングで切り落とされて読み出される。例えば <code>var="/a/b/c"; echo "\${var%/*}"</code> の場合、“a/b” が読み出される。
<code>\${var%%pattern}</code>	変数 <code>\$var</code> に格納されている文字列の右端に、パターン “pattern” があれば、それが、最大マッチングで切り落とされて読み出される。例えば <code>var="/a/b/c"; echo "\${var%%/*}"</code> の場合、“”（空文字）が読み出される。

いずれも、条件に該当しない場合は元のシェル変数に格納されている文字列が読み出される。

レシピ 5.3 スコープ

→レシピ 5.11（local 修飾子）を参照

レシピ 5.4 正規表現

正規表現に関して注意しなければならないことは、コマンドの種類（AWK、grep、sed など）や、また同じコマンドでも環境によって使えるメタ文字の範囲が異なるということだ。詳細に関しては本章第二部（正規表現）を参照してもらいたい。

ちなみに、「シェル変数の正規表現は？」と質問する人がいるかもしれないが、それは一部シェルの独自拡張

機能なのでどの環境でも使えるものではない。

→レシピ 5.7 (ロケール)、レシピ 5.5 (文字クラス) も参照

レシピ 5.5 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。

この正式名称は「POSIX 文字クラス」*1 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。

まあ、POSIX に準拠してないそっちの実装が悪いといってしまうまでもなのだが、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。

→レシピ 2.3 (全角文字に対する正規表現の扱い) も参照

レシピ 5.6 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 `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。

```
$ od -A n -t u4 -N 4 /dev/urandom | sed 's/[^0-9]//g' ↵
```

*1 使えるもの一覧は、http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html#tag_09_03_05 参照

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 5.33 (`tr` コマンド) を参照。

/dev/urandom をどうしても使いたくない場合

乱数の品質は `/dev/urandom` ほど高くないものの、代替手段はある。`ps` コマンドの結果は実行するたびに必ず変化するのでこれを種として取り入れる。

具体的には、プロセス ID、実行時間、CPU 使用率、メモリ使用量の各一覧あたりが刻々と変化するのでこれらを取得するとよいだろう。更に、現在日時も加え、これらに基づいて 2^{32} 未満の範囲で `AWK` の `srand()` に渡す乱数の種を生成しているのが次のコードだ。

```
LF=$(printf '%Yn_');LF=${LF%_}      # sed で改行を扱うための定義
(ps -Ao pid,etime,pcpu,vsz; date) | # 乱数源 (プロセス情報一覧 + 日時)
od -t d4 -A n -v                    | # 数値化する
sed 's/[^0-9]¥{1,¥}/'"$LF"'/g'      |
grep '[0-9]'                         |
tail -n 42                           | # 100000000 未満の数字を
sed 's/.*¥(.¥{8¥}¥)¥/¥1/g'          | # 42 個まで用意 (2^32 未満にするため)
awk 'BEGIN{a=-2147483648;}           # # 上の値を足して signed long 値を作る
     {a+=¥1;}                        #
     END{srand(a);print rand();}'
```

レシピ 5.7 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。

具体的に何がかわるかといえば、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もの

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 及び LANG=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 LANG=C awk '{print length($0)}'
```

■予め LC_ALL=C 及び LANG=C を設定しておく

```
export LC_ALL=C LANG=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
```

ロケール系環境変数には現在、LANGUAGE と LC_* と LANG がある。このうち各種 LC_* については LC_ALL の設定によって全て上書きされるが、LANG には効かないので LC_ALL と LANG を両方とも “C” にする。最初に列挙した LANGUAGE は最も強い効力を持つようだが、LANG や LC_ALL に “C” が設定されている場合は無視されるということである*2。

ちなみに、いにしへの export は、= を使って変数の定義と export 化を同時に行えなかったということだが、今どきの POSIX の man ページ*3によれば使えることになっている。

→レシピ 5.1 (環境変数等の初期化)、レシピ 2.4 (sort コマンドの基本と応用とワナ) も参照

*2 GNU gettext ドキュメント 2.3.3 項

http://www.gnu.org/software/gettext/manual/html_node/The-LANGUAGE-variable.html#The-LANGUAGE-variable

*3 http://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html#export

レシピ 5.8 \$(式)

よく「計算をさせたいければ 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 5.14 (AWK コマンド) 参照

レシピ 5.9 case 文

→レシピ 5.10 (if 文) 参照

レシピ 5.10 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。

■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 5.11 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。(ただし初期値はそれら外部変数の値になっている)

■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 5.12 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る、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 正規表現

正規表現は、コマンドによって使えるメタ文字の範囲が異なるうえ、実装によっては POSIX の範囲で使えるメタ文字の他に独自のメタ文字を追加していることもある。おかげで、結局どのコマンドでどのメタ文字が使えるのか (使ってもよいのか) わからず混乱しがちだ。そこで、POSIX の範囲を意識しながら「どの UNIX コマンドでも使える正規表現」をまとめた。

知っておくべきメタ文字セットは3つ

知っておくべきメタ文字セットは、次に示す2種類+1サブセットの3つだけである。

1. BRE（基本正規表現）メタ文字セット^{*4}
2. ERE（拡張正規表現）メタ文字セット^{*5}
 - 2/ AWK のサブセット

もちろん、これ以外にも GNU 拡張正規表現メタ文字セットや Perl 拡張正規表現メタ文字セット、JavaScript 拡張正規表現メタ文字セットなどいくつかあるのだが、UNIX における「どの環境でも（=POSIX で）使える」という特長を持たせたいのであれば、それらを意識する必要はなく、この3つさえおさえておけばよい^{*6}。

各コマンドは、どのメタ文字セットに対応しているか

知っておくべきメタ文字セットが3つあることがわかったところで、各コマンドがそれら3つのうちどれに対応しているかをまとめたのが次の表だ。この表を見て、対応しているメタ文字セットがわかったら、次項のメタ文字セット各一覧を見ればよい。

表 5.2 各コマンドが対応しているメタ文字セット

コマンド	対応しているメタ文字セット
AWK	ERE の AWK サブセット
ed	BRE メタ文字セット
egrep	BRE メタ文字セット
ex	BRE メタ文字セット
grep(-E なし)	BRE メタ文字セット
grep(-E あり)	ERE メタ文字セット
more	BRE メタ文字セット
sed	BRE メタ文字セット
vi	BRE メタ文字セット

参考までに述べておくと、**GNU 拡張**や **Perl 拡張**、**JavaScript 拡張**は、いずれも **ERE** のスーパーセットである。ということはすなわち、ERE メタ文字セットを覚えておけばそれらの上でも動くということだ。

また、多くの grep（-E オプションなし）では、一部の ERE メタ文字がバックスラッシュ付きで使えたりする（例えば “¥+” や “¥|”）が、それらは GNU 拡張であって grep 本来のものではない。

目的のコマンドがどのメタ文字セットに対応しているかわかったところで、次項より各メタ文字セットを紹介する。

^{*4} 原文は POSIX の 9.3 節 “Basic Regular Expression” 参照。

(http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html#tag_09_03)

^{*5} 原文は POSIX の 9.4 節 “Extended Regular Expression” 参照。

(http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html#tag_09_04)

^{*6} 例え POSIX にこだわらないとしても、他のものは大抵 ERE の拡張になっているので、一旦 ERE を覚えておくと整理しやすいだろう。

1. BRE（基本正規表現）メタ文字セット

（BREに限らないが）メタ文字セットの中は、メタ文字を使う場所に依じてさらに3つのグループに分類される。それを踏まえて読んでもらいたい。

a-1. マッチを掛ける文字列（置換前文字列）のメタ文字一覧（ブラケット外部）

まずはブラケット（`[~]`）の外部についてのみまとめる。ブラケット内部ではここで記す多くのメタ文字が意味を失ったり、あるいは意味が変わったり、ブラケット内部でのみ意味を持つメタ文字が新たに登場するため、次の表でまとめることにする。

メタ文字	意味
<code>^</code>	文字列（通常は行）の先頭にマッチ（先頭以外では通常文字と見なされる）
<code>\$</code>	文字列（通常は行）の末尾にマッチ（末尾以外では通常文字と見なされる）
<code>[...]</code>	<code>[と]</code> で囲まれた中で列挙した文字のいずれか1文字にマッチ
<code>[^...]</code>	<code>[^と]</code> で囲まれた中で列挙した文字以外の任意の1文字にマッチ
<code>*</code>	【繰返し指定子】 直前に記述した文字が0文字以上連続していることを指定し、後続の繰返し指定子よりも優先して可能な限り最大数マッチさせようとする。
<code>¥{n¥}</code>	【繰返し指定子】 直前に記述した文字が n 文字連続していることを指定する。
<code>¥{n,¥}</code>	【繰返し指定子】 直前に記述した文字が n 文字以上連続していることを指定し、後続の繰返し指定子よりも優先して可能な限り最大数マッチさせようとする。
<code>¥{m,n¥}</code>	【繰返し指定子】 直前に記述した文字が m 文字以上、 n 文字以下連続していることを指定し、後続の繰返し指定子よりも優先して可能な限り最大数マッチさせようとする。
<code>¥(...¥)</code>	【包括指定子】 <code>¥(と¥)</code> で囲まれた範囲の文字列を、上記の繰返し指定子の1文字として扱わせたい場合、もしくは <code>sed</code> 等で置換後に再利用したい文字列範囲を指定したい場合に用いる。
<code>¥n</code>	【後方参照子】 n 番目に記した包括指定子でマッチした文字列にマッチする。例えば、 <code>ABC123ABCABC</code> という文字列を <code>^¥([A-Z]*¥)123¥1*\$</code> という正規表現文字列に掛ければ、 <code>¥1</code> は <code>ABC</code> という文字列と見なされるため、この場合 <code>¥1*</code> は末尾にある2つの <code>ABC</code> にマッチする。
<code>¥x</code>	上記のうちでバックスラッシュで始まらないメタ文字自身、あるいは <code>AWK</code> や <code>sed</code> などで正規表現の始まりを示すために用いた文字自身を指定したい場合に、 x の部分にその文字を記述すればそれにマッチ
<code>¥¥</code>	バックスラッシュ（ <code>¥</code> ）自身にマッチ

a-2. マッチを掛ける文字列（置換前文字列）のメタ文字一覧（ブラケット内部）

既に述べたように、ブラケット（`[~]`）で囲まれた区間は外側とは使えるメタ文字が異なり、次の表のとおりである。

メタ文字	意味
~	開きブラケットの直後 (“-”や “[” 自身を指定したい場合でもそれらより手前) に記述すると、否定の意味になる。
-	文字を列挙する代わりに範囲で指定できる。例えば A-Z ならば文字 A から Z を全て列挙したと同等。もし “-” 自身を指定したい場合は、閉じブラケットの直前 (閉じブラケット “[” 自身も指定したい場合はそちらよりも後ろ) に記述する。
[閉じブラケット (]) の直前 (ただし “-” 自身も指定する場合はそちらの方が後ろ) に記述すると “[” 自身を指定することができる。
]	開きブラケット ([) の直後に記述すると “]” 自身を指定することができる。
¥ <i>x</i>	上記のうちでバックスラッシュで始まらないメタ文字自身、あるいは AWK や sed など正規表現の始まりを示すために用いた文字自身を指定したい場合に、 <i>x</i> の部分にその文字を記述すればそれにマッチ
¥¥	バックスラッシュ (¥) 自身にマッチ

実は BRE (後述の ERE も含む) では、上記に加えて次の表に示すメタ文字列が定義されているのだが、間違った実装がされていたり、使える実装にお目にかかったことがなかったり、といったものであるため、使うことはお勧めできない。

メタ文字	意味
[<i>:word:</i>]	【POSIX 文字クラス】 <i>word</i> の部分には alnum (アルファベットと数字全部)、 cntrl (制御文字全部)、 lower (アルファベット小文字全部)、 space (スペースとタブと改ページ)、 alpha (アルファベット全部)、 digit (数字全部)、 print (制御文字以外の文字全部)、 upper (アルファベット大文字全部)、 blank (スペースとタブ)、 graph (制御文字とスペース、タブ以外全部)、 punct (句読点全部)、 xdigit (16 進数文字全部) が指定できる。実際に使うときは [[:lower:][:blank:]] などのように使う。しかし、一部の实装ではブラケット記号が一重でないと動かないといった間違った実装になっているものがある。
[<i>.word.</i>]	例えば [[.hoge.]] と記述したら、¥(hoge¥)¥{1,¥} と等価な意味を持つようだ。しかし、使える実装を見たことがない。後者の記述で事足りるからだろうか？
[<i>=x=</i>]	例えば [=a=] と記述したら、“a” にも “â” にも “ã” にもマッチするもので、実際に使う時は [[=a=]bc] のように記述する。だがこれも、使える実装を見たことがない。アクセント記号付の文字を素直に書き並べれば済むので無くても事足りるからだろうか？

b. 置換後の文字列指定 (sed 等の “s/A/B/” における B の部分) で使えるメタ文字一覧

正規表現は、マッチする文字列を検索するためだけでなく、マッチしたその文字列を加工 (置換) するためにも用いられる。sed コマンドにおける “s/A/B/” はそのための代表的な書式であるが、この B の部分で使えるメタ文字を次の表に示す。

メタ文字	意味
<code>¥n</code>	<code>n</code> 番目に記した包括指定子 (<code>¥(…¥)</code>) で囲まれた範囲にマッチした文字列に置き換えられる。
<code>&</code>	マッチした文字列全体に置き換えられる。
<code>¥x</code>	上記のメタ文字 (<code>&</code>)、また <code>sed</code> 等で正規表現の始まりを示すために用いた文字自身を指定した場合、 <code>x</code> の部分に記せばその文字自身を指定することができる。
<code>¥¥</code>	バックスラッシュ (<code>¥</code>) 自身を指定したい場合に用いる。

2. ERE（拡張正規表現）メタ文字セット

ERE はほぼ BRE を拡張したものになっているが、純粋の上位互換ではないので注意。ERE のメタ文字一覧も表で示すが、予め違いを簡単に列挙すると次のとおりである。

- 使えるメタ文字を追加。（`+`、`?`、`|`）
- 【非互換】 バックスラッシュでエスケープしていた括弧類（`¥(`、`¥)`、`¥{`、`¥}`）がバックスラッシュ不要に。
- 【非互換】 後方参照（`¥n`）が無保証に。（実際に使えない実装がある）

使えるメタ文字が増えている（`+`、`?`、`|`）が、純粋な上位互換ではないので注意。具体的には、バックスラッシュでエスケープしていた括弧類のメタ文字（`,`、`{`、`}`）がバックスラッシュ不要になっている点、そして後方参照が保証されていない（実際に使えない実装がある）点である。

a-1. マッチを掛ける文字列（置換前文字列）のメタ文字一覧（ブラケット外部）

BRE と同様に、まずマッチを掛ける文字列（置換前文字列）として指定できるメタ文字をまとめると、次の表のとおりになる。

メタ文字	意味
<code>^</code>	文字列（通常は行）の先頭にマッチ（先頭以外では通常文字と見なされる）
<code>\$</code>	文字列（通常は行）の末尾にマッチ（末尾以外では通常文字と見なされる）
<code>[...] </code>	<code>[と]</code> で囲まれた中で列挙した文字のいずれか 1 文字にマッチ
<code>[^...] </code>	<code>[^と]</code> で囲まれた中で列挙した文字以外の任意の 1 文字にマッチ
<code>*</code>	【繰返し指定子】 直前に記述した文字が 0 文字以上連続していることを指定し、後続の繰返し指定子よりも優先して可能な限り最大数マッチさせようとする。
<code>+</code>	【繰返し指定子】 直前に記述した文字が 1 文字以上連続していることを指定し、後続の繰返し指定子よりも優先して可能な限り最大数マッチさせようとする。
<code>{n}</code>	【繰返し指定子】 直前に記述した文字が n 文字連続していることを指定する。
<code>{n,}</code>	【繰返し指定子】 直前に記述した文字が n 文字以上連続していることを指定し、後続の繰返し指定子よりも優先して可能な限り最大数マッチさせようとする。
<code>{m,n}</code>	【繰返し指定子】 直前に記述した文字が m 文字以上、 n 文字以下連続していることを指定し、後続の繰返し指定子よりも優先して可能な限り最大数マッチさせようとする。
<code>?</code>	【繰返し指定子】 直前に記述した文字が 0 文字以上 1 文字以下連続していることを指定する。
<code>(...)</code>	【包括指定子】 <code>(と)</code> で囲まれた範囲の文字列を、上記の繰返し指定子の 1 文字として扱わせたい場合、もしくは <code>sed</code> 等で置換後に再利用したい文字列範囲を指定したい場合に用いる。または、後述の論理和指定子の範囲を限定したい場合に用いる。
<code> </code>	【論理和指定子】 この指定子の左の文字列または右の文字列でマッチさせることを指定する。左右の範囲は、前述の包括指定子の中であればその始端または終端まで、無ければ正規表現文字列全体の始端または終端まで（ <code>^</code> や <code>\$</code> をも内包させる）と見なされる。例えば <code>^ABC DEF\$</code> は、 <code>^(ABC DEF)\$</code> ではなく <code>(^ABC) (DEF\$)</code> の意味に解釈される。
<code>¥x</code>	上記のうちでバックスラッシュで始まらないメタ文字自身、あるいは <code>AWK</code> や <code>sed</code> などで正規表現の始まりを示すために用いた文字自身を指定したい場合に、 x の部分にその文字を記述すればそれにマッチ
<code>¥¥</code>	バックスラッシュ (<code>¥</code>) 自身にマッチ

a-2. マッチを掛ける文字列（置換前文字列）のメタ文字一覧（ブラケット内部）

これは BRE と同じ。

b. 置換後の文字列指定で使えるメタ文字一覧

これも BRE と同じ。たが置換後文字列を指定できるコマンドで ERE に対応しているものは、POSIX の範囲では存在しない。（`AWK` は後述するのでここでは除く）

2f. AWK で使えるメタ文字セット

`AWK` は基本的には ERE のメタ文字セットに対応しているが、残念なことにブレース（`{`、`}`）には対応していない。従ってブレースによる繰返し指定はできず、大きな弱点になっている。

一応 2008 年版の POSIX ではこれも含めて ERE に完全対応するように勧告されたようだが、まだ年数が浅

いたために現存する AWK 実装で対応しているものは少なく、実質的に完全な ERE は通用しない。

また AWK では、正規表現を制御する各構文や関数に正規表現文字列が渡される前に、AWK 言語としてのエスケープ処理がなされるので注意が必要だ。具体的には、次のとおりである。

文字	何の文字に置換されるか
<code>\</code>	バックスラッシュ <code>\</code> に置換される。
<code>/</code>	スラッシュ <code>/</code> に置換される。
<code>"</code>	ダブルクォーテーション <code>"</code> に置換される。
<code>ddd</code>	<code>ddd</code> が 3 桁の 8 進数である時、その値の文字コードに該当する文字に置換される。
<code>a</code>	ビーブ音 (BEL:文字コード 0x07) に置換される。
<code>b</code>	バックスペース (BS:文字コード 0x08) に置換される。
<code>f</code>	改ページ (FF:文字コード 0x0c) に置換される。
<code>n</code>	改行 (LF:文字コード 0x0a) に置換される。
<code>r</code>	行頭復帰 (CR:文字コード 0x0d) に置換される。
<code>t</code>	水平タブ (HT:文字コード 0x09) に置換される。
<code>v</code>	垂直タブ (VT:文字コード 0x0b) に置換される。
<code>(上記以外)</code>	未定義 (通常は単にバックスラッシュを除いた文字に置換される)

a-1. マッチを掛ける文字列 (置換前文字列) のメタ文字一覧 (ブラケット外部)

ブレースを用いた繰返し指定子 (“{”、“}”) 以外の全ての ERE メタ文字セットに対応している。ただし、バックスラッシュで始まる文字列を与えると先程の表のとおりのエスケープ処理を受ける。

a-2. マッチを掛ける文字列 (置換前文字列) のメタ文字一覧 (ブラケット内部)

BRE と同じ。ただし、バックスラッシュで始まる文字列を与えると先程の表のとおりのエスケープ処理を受ける。

b. 置換後の文字列指定 (sub, gsub 関数の第 2 引数) で使えるメタ文字一覧

これも基本的には BRE と同じなのだが、次の 2 点に注意しなければならない。

- “`\n`” (n は自然数) というメタ文字には対応していない (これに対応しているのは GNU AWK で追加された `gensub` という関数である)。
- 先程の表のとおりのエスケープ処理を受ける。このため、特に注意が必要なのは、メタ文字として予約されている “`&`” 自身を指定したい場合である。具体的には、ダブルクォーテーションの内側で “`\&`” と書きたいのであれば、バックスラッシュ “`\`” がエスケープされないように “`\\&`” と記されなければならない。

■ 第三部 コマンド

いくらシェルスクリプトの文法、また正規表現に気をつけても、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、POSIX の範囲で使用可能なオプションや、同じ使い方をしてい

るにも関わらず生じる動作の違い、そして対策、といったことを中心に、どの環境でも通用するコマンド使用法の各論を紹介する。

レシピ 5.13 “[” コマンド

→レシピ 5.32 (test コマンド) 参照

レシピ 5.14 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。

マイナスを取り去るには

計算で得られた結果に 0 を足せばよいようだ。

```
$ awk 'BEGIN{print -1*0+0}'  
0  
$
```

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 `arlen()` 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() 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 5.26 (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 5.6 (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 関数がその一つであることだ。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 関数を使え。その他、こまごまと気を付けるべきことについては、GNU AWK マニュアル “--posix” オプションに関する記述^{*7}が参考になる。

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……。

AWK で使える正規表現のメタ文字に関しては、本章第二部 (正規表現) で詳しく紹介しているので参照してもらいたい。

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？

^{*7} http://www.gnu.org/software/gawk/manual/gawk.html#index-gawk_002c-extensions_002c-disabling

```
$ awk 'BEGIN{print 2147483648}'  
2.14748e+09
```

上記の例は、0x7FFFFFFF（符号付き 4 バイト整数の最大値）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、単に表示させただけなら文字列として扱えばよい。

ロケール

→レシピ 5.7（ロケール）を参照

レシピ 5.15 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）
- “+ 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^{*8}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^{*9}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 5.16 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

^{*8} <http://pubs.opengroup.org/onlinepubs/9699919799/utilities/date.html>

^{*9} 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 ↵
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

```
$ 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あアリ
```

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 5.17 echo コマンド

結論から言うと、どこでも動くようにしたい場合、次の項目に 1 つでも当てはまるものには **echo** コマンドは使うべきではない。

- 先頭がハイフンで始まる可能性がある文字列
- エスケープシーケンスを含む可能性のある文字列

理由は次のとおりである。

対応しているオプションが異なる

例えば、Linux の echo コマンドは **-e**、**-n** オプションに対応しており、第一引数に指定すれば、それを表示はせずにオプション文字列として解釈する。一方、FreeBSD の echo コマンド（外部コマンド版）は **-n** オプションのみに対応しており、第一引数に “**-e**” を与えれば表示する。また一方、AIX の echo コマンドはどちらにも対応していないため、第一引数に “**-e**” や “**-n**” を与えるとどちらも表示する。このように対応がバラバラだからだ。

エスケープシーケンスに反応する実装がある

例えば **¥n** は改行を意味するエスケープシーケンスであるが、FreeBSD の echo はそのまま “**¥n**” と表示する。一方、Linux の echo は **-e** オプションが付けられた時のみ改行に置換される。また一方、AIX の echo は常に改行に置換する。

AIX の echo はデフォルトでエスケープシーケンスを解釈するのだ。「それ POSIX 的になの？」と困惑するかもしれないが、POSIX の echo の man^{*10}にはちゃんとエスケープシーケンスの記述がある。

対策

どんな文字列が入っているかわからない変数を扱う場合（ハイフンで始まらないとかエスケープシーケンスを含まないとわかっているならそのままよい）、例えば次のように printf コマンドを使うなどして回避すること。

■echo のオプション反応問題を回避する例（printf で代用）

```
#!/bin/sh
for arg in "$@"; do
    printf '%s¥n' "$arg"
done
```

レシピ 5.18 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。

^{*10} <http://pubs.opengroup.org/onlinepubs/9699919799/utilities/echo.html>

```
$ 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 ↵  
$ exec 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 5.19 fold コマンド

一般的に、ファイル名として “-” を指定すると標準入力の意味と解釈されるが、本コマンドに対しては使わない方がよい。POSIX には fold コマンドでも、“-” は標準入力だと解釈されると確かに書いてあるのだが、BSD の実装では真面目に “-” というファイルを開こうとしてエラーになってしまう。

レシピ 5.20 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 で使える正規表現メタ文字セットは、`-E` オプション無しの場合には BRE（基本正規表現）だけ。`-E` オプション付きの場合には ERE（拡張正規表現）で規定されているものだけだ。詳しくは、本章第二部（正規表現）を参照のこと。

レシピ 5.21 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^{*11}。

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 3 バイトとして、それを 1 つ（count）と指定すればよい。

```
$ echo 12345 | dd bs=3 count=1 2>/dev/null ↵  
123$
```

これは標準入力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=3 count=1 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

^{*11} AIX では最後に余計な改行コードが付く。

に `2>/dev/null` などと書いて、ログを捨てること。

レシピ 5.22 ifconfig コマンド

これも POSIX で規定されていないコマンドだし、最近の Linux では使われ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。

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 1.8 (IP アドレスを調べる (IPv6 も)) 参照

レシピ 5.23 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^{*12}によれば、どの環境でも使える番号は 5.3 に記したものの以外保証されていない。

表 5.3 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。

^{*12} <http://pubs.opengroup.org/onlinepubs/9699919799/utilities/kill.html>

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 の `signal.h` に関する項^{*13}にも記されているとおり、豊富にある。

-1 オプションは避ける

`kill` コマンドで `-1` 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 5.24 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。

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。簡易的な対処と本格的な対処の 2 種類を用意した。

簡易的な mktemp

一意性のみでセキュリティは保証しない簡易的なもの^{*14}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

```
type mktemp >/dev/null 2>&1 || {
  mktemp_fileno=0
  mktemp() {
    (
      filename="/tmp/${0##*/}.${$.}.$mktemp_fileno"
      touch "$filename"
      chmod 600 "$filename"
      printf '%s\n' "$filename"
    )
    mktemp_fileno=$((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。

本格的な mktemp

POSIX 版 `mktemp` コマンド^{*15}を作ってしまったので、これをダウンロードして使えばよい。

^{*13} <http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/signal.h.html>

^{*14} もしセキュリティを確保したい場合は良質な乱数源が必要となる。→レシピ 5.6（乱数）参照

^{*15} <https://github.com/ShellShoccar-jpn/misc-tools/blob/master/mktemp>

書式は Coreutils 版^{*16}に似せてある。ただし動作パフォーマンス確保のため、`/bin` や `/usr/bin` の中に元々の `mktemp` が存在すればそちらを使う（`exec` する）ようにしてあるので、あまり一般的でないオプションは使わない方がよい。

レシピ 5.25 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！^{*17}）

`-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。あふれた場合の動作は POSIX で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。

■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

^{*16} https://www.gnu.org/software/coreutils/manual/html_node/mktemp-invocation.html#mktemp-invocation

^{*17} POSIX の範囲ではないのだが、`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。

```
$ 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。

対応方法

AWK コマンドの組み込み変数である `NR` を使うとよい。さらに、次のようにして `printf` 関数を併用すれば、GNU 版 `nl` コマンドと同等の動作をする。

■GNU 版 `nl` コマンドのデフォルトと同じ動作をする

```
awk '{printf("%6d¥t%s¥n",NR,$0);}'
```

レシピ 5.26 printf コマンド

キャラクターコードによる即値指定

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。

負の 16 進数

負の値を 16 進数に変換すると環境によって結果が異なる。例えば `-1` を 16 進数に変換すると次のとおりだ。

■32 ビット実装の場合

```
$ printf '%X\n'-1 ↵
FFFFFFF
$
```

■64 ビット実装の場合

```
$ printf '%X\n'-1 ↵
FFFFFFFFFFFFFFFF
$
```

従って負の値を 16 進数に変換するのはあまり勧められないが、どうしてもしたいなら下 8 桁のみを取り出すべきだろう。もちろんその場合、-2147483648 より小さい値は扱えない。

レシピ 5.27 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が厄介だ。

-x オプションは避ける

-x オプションは「制御端末を持たないプロセスを含める」という働きを持つが、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^{*18}には記載されていないし、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、何のオプションも付けずに自分のプロセスのみを表示した場合にある GNU 版で表示されたプロセスが、BSD 版では -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、

^{*18} <http://pubs.opengroup.org/onlinepubs/9699919799/utilities/ps.html>

- 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)

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 `init` 以外のシステムプロセスの親プロセス ID 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 5.28 readlink コマンド

このコマンドは、与えられた引数がシンボリックリンクだった場合にその実体のパスを教えてくれるという便利なコマンドだ。しかし、残念ながら POSIX のコマンド群には存在しない。

直接的な答えではないが、シンボリックリンクの実体を求めるシェルスクリプトのコードをレシピ 1.5（一時ファイルを作らずファイルを更新する）で紹介しているので参考にしてもらいたい。

レシピ 5.29 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 1.7（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 コマンドマニュアル^{*19}を見る。また、sed が対応している正規表現メタ文字セットは BRE（基本正規表現）であり、本章第二部（正規表現）にその一覧を記してあるので参照してもらいたい。

標準入力指定の “-”

一般的に、ファイル名として “-” を指定すると標準入力の意味と解釈されるが、本コマンドに対しては使わない方がよい。POSIX には sed コマンドでも、“-” は標準入力だと解釈されると確かに書いてあるのだが、BSD の実装では真面目に “-” というファイルを開こうとしてエラーになってしまう。

ロケール

→レシピ 5.7（ロケール）を参照

レシピ 5.30 sort コマンド

→レシピ 5.7（ロケール）を参照

レシピ 5.31 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。

^{*19} <http://pubs.opengroup.org/onlinepubs/9699919799/utilities/sed.html>

■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 -k 1nr,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。^{*20}

■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 5.32 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^{*21}。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。

^{*20} bsdhack 氏のブログ記事 <http://blog.bsdhack.org/index.cgi/Computer/20100513.html> より

^{*21}さすがに\$str の中身が “(” だった場合ダメなようだが。

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1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いる場合もあるが）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
```

レシピ 5.33 tr コマンド

このコマンドは各 UNIX の系譜に基づく方言が強く残るコマンドの一種で、どこでも動くプログラムを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 System V 系の書式にある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 -d 'a-z' ← BSD 系、POSIX での書式
```

POSIX に準拠していない 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'
```

しかしながら、改行コードで終わっていないテキストデータを与えると改行を付け足してしまう sed 実装があるので、そういう可能性のあるデータを扱いたい場合は更に対策が必要だ。→レシピ 1.7（改行無し終端テキストを扱う）を参照

そこまでやるくらいだったら、範囲指定ではなく全部書いてしまえばいいと思うかもしれないが、もちろんそれでもいい。

```
tr -d 'abcdefghijklmnopqrstuvwxyz'
```

レシピ 5.34 trap コマンド

→レシピ 5.23 (kill コマンド) 参照

レシピ 5.35 which コマンド

コマンドが存在すれば（パスが通っていれば）そのパスを返してくれるため、コマンドが無ければ無いなりなどの環境でも動くようなシェルスクリプトを書いた時などに重宝するコマンドだ。ところが、この which コマンドが POSIX 標準ではないというオチが待っている。

しかし諦めることはない。POSIX のコマンドで似た働きをする type というものがある。ただこれは人間向けの文章で出力されるので、それを解析してパスを取り出す必要がある。次のコードをシェルスクリプトの冒頭に追記しておけば、which コマンドが存在しない場合のみ、type コマンドに基づいたシェル関数版 which が登録される。

■which コマンドが無ければ同等品を追加するコード

```
which which >/dev/null 2>&1 || {
  which () {
    (
      ans=$(LANG=C LC_ALL=C type "$1" 2>/dev/null) || exit $?
      case "$1" in */*) printf '%s\n' "$1"           ; exit;; esac
      case "$ans" in */*) printf '%s\n' "${ans#*/}"; exit;; esac
      printf '%s\n' "$1"
    )
  }
}
```

レシピ 5.36 xargs コマンド

改行なし終端データの扱い

次の例を見てもらいたい。

```
$ printf 'one two threee' | xargs echo
one two
$
```

単語が3つあるのだから、xargs は echo の後ろに “one” と “two” はもちろん、“three” も付けて実行してくれることを期待するが、最後の “three” が無視されてしまっている。じつはこの xargs 実装、最後の単語の後も改行やスペース等の列区切り文字を必要とするのである。

こういう xargs 実装であっても確実に動作させるようにするには、例えば xargs の直前に grep ^などを挿んでデータの終端に確実に改行が付くようにしてやることだ。

```
$ printf 'one two threee'| grep ^ | xargs echo ↵
one two three
$
```

空ループの有無

標準入力から入ってきた文字列を引数にしてコマンドに渡すためのコマンドであるが、標準入力から空白以外が含まれた行が1行も渡ってこなかった場合、引数無しでコマンドを実行する xargs 実装もあれば、コマンドを実行しない xargs 実装もある。

■BSD 版の場合

```
$ printf ' %n%n'| xargs echo 'foo' ↵
$
```

■GNU 版 (多くの Linux) の場合

```
$ printf ' %n%n'| xargs echo 'foo' ↵
foo
$
```

xargs で呼び出される側のコマンドは引数 0 個で呼ばれるなど想定していない (Usage を表示したり戻り値 0 以外にしたりする) ものが多いので、前者の挙動の方が好ましいとは思うのだが、あるものはしょうがない。

一応、前者の動作に揃える `-r` オプションというものがある (最近の FreeBSD 版もこれを認識する) のだが、そんなオプションは POSIX では規定されていないがゆえ、それを付けて互換性を向上させようとする逆にと逆全ての環境で動く保証がなくなってしまうのが皮肉なところ。

対応方法

さてどうするか……、これは対症療法しかない。すなわち、

1. 引数 0 個で実行されてもエラー扱いしないようなコマンドにする。
2. コマンドがエラー動作することを想定するような後続の処理にする。
3. 標準入力に必ず有効かつ無害な行が入るようにする。
4. 呼び出されるコマンドに無害な引数を付けておく。

などを行う。

1 番目の対処は、例えば呼び出すコマンドが `rm` なら `-f` オプションを付けてエラー扱いを抑止するという方法だ。

■対処方法 1 の例「rm コマンドをいちいちエラーで騒がせないようにする」

```
find . -name '*.tmp' | xargs rm -f
```

2 番目の対処は、例えば戻り値が 0 以外でも即エラー扱いしないとか、標準エラーに流れてくるエラーメッセージや Usage を /dev/null に捨てるといったものだ。

■対処方法 2 の例「rm コマンドがエラーで騒いでも無視する」

```
(find . -name '*.tmp' | xargs rm) 2>/dev/null
```

3 番目の対処は、例えば呼び出すコマンドが grep などのファイルを読み込むだけのものであれば使える方法だ。例えば /dev/null を読み出しファイルとして、標準入力の最初（最後に付加すると改行なし終端テキストだった場合に不具合が起こる）に付加すればよい。

■対処方法 3 の例「grep に無害なファイル /dev/null を読み込ませる」

```
# grep の場合は後述の 4 番目の対処方法をお勧めする
find . -name '*.txt' | awk 'BEGIN{print "/dev/null"} 1' | xargs grep '検索キーワード'
```

4 番目の対処は、手段が若干異なるだけで目的は 3 番目と同じだ。先程の grep の例ならこう書き直す。短く書けるし、先程紹介した対処方法よりもお勧めだ。

■対処方法 4 の例「grep に無害なファイル /dev/null を読み込ませる（推奨）」

```
find . -name '*.txt' | xargs grep '検索キーワード' /dev/null
```

grep コマンドの場合は特にこちらを勧める。理由は、grep コマンドは、検索対象のファイルが 1 個だけ指定された場合と複数指定された場合で挙動を変えるからだ。具体的には、検索キーワードが見つかった時、1 個だけだった場合はファイル名を表示しないのに対し、複数個だった場合には行頭にファイル名を併記する。

上記のように記述しておけば、grep コマンドは常に複数個指定されたと見なすので、find コマンドで見つかったファイルの数が 1 個であっても 2 個以上であっても、必ず行頭にファイル名を併記するようになり、動作が統一される。

引数文字列の扱い

xargs に ~~¥¥¥~~ という文字列を与えると、例えば FreeBSD の xargs と Linux の xargs で異なった結果を返す。

■FreeBSD の場合

```
$ printf '¥¥¥¥¥' | xargs printf
,
$
```

■GNU 版 (多くの Linux) の場合

```
$ printf '¥¥¥¥¥' | xargs printf
¥'
$
```

実は FreeBSD の xargs コマンドは、引数文字列を \$@ (ダブルクォーテーションなし) のように渡してシェルのエスケープ処理を受けるのに対し、Linux の (GNU 版の) xargs コマンドは "\$@" (ダブルクォーテーション

あり)のように渡すので、シェルのエスケープ処理を受けない。だから結果として、Linux 上ではバックスラッシュが1個残るのだ。

ではどうするか。確実な方法は、エスケープ処理される文字を使わないことだ。バックスラッシュはいたしかたないとして、例えばシングルクォーテーションは¥047 などと表現した文字列が printf に渡るようにすればよい。ただしバックスラッシュも、引数としてシェルに解釈される時や printf に解釈される時などにエスケープ処理を受けるので十分注意すること。

レシピ 5.37 zcat コマンド

zcat は、gunzip | cat 相当だと思っている人も多いかもしれないが違う！それは GNU 拡張であり、本来の zcat は uncompress | cat 相当である。

従って、次のようにして gzip 圧縮されたデータを与えるとエラーを返す zcat コマンド実装がある。

```
$ echo hoge | gzip | zcat ↵
stdin: not in compressed format
$
```

全ての環境の zcat コマンドを想定するなら、compress コマンドで圧縮したデータを与えること。

```
$ echo hoge | compress | zcat ↵
hoge
$
```

ファイルを経由する場合の注意点

compress コマンドは元データがファイルの場合、圧縮してもサイズが小さくならないと判明すると圧縮をしないという性質がある。そのため、普通に使うと次のような事故が起きる恐れがある。

```
$ echo 1 > hoge.txt ↵
$ compress hoge.txt ↵
--file unchanged ← サイズが小さくならないので圧縮ファイルは作られなかった
$ zcat hoge.txt.Z ↵
hoge.txt.Z: No such file or directory
$
```

これを防ぐためには、-f オプションを付ければよい (compress -f とする)。

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……、都道府県名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。次の表をご覧ください。

^{*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）
| |
| |   ・ unzip コマンド、curl コマンド、及び iconv または nkf コマンド
| |   ・ cron 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、日本郵便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行くのは効率が悪い。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な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で 2 種類のデータ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、該当する住所等の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 4.7（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 形式で返すこともできる」ということを示すためだ。JSON 形式で返せば、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、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 -----
type curl >/dev/null 2>&1
[ $? -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 's/_/_/' | 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 's/_/_/' | 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 type iconv >/dev/null 2>&1; then #
    iconv -c -f SHIFT_JIS -t UTF-8 #
elif type nkf >/dev/null 2>&1; 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">

<head>
<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>
</head>

<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
# 郵便番号一住所検索
#
# [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=="'"$zipcode"'"{hit=1;print;exit} END{if(hit==0){print ""}}' | # 該当行を取り出し (1行のみ)
while read zip pref city town; do # HTTP ヘッダーと共に、JSON 文字列化した住所データを出力する
    cat <<__HTTP_RESPONSE
        Content-Type: application/json; charset=utf-8
        Cache-Control: private, no-store, no-cache, must-revalidate
        Pragma: no-cache
        {"zip":"$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}も用意しているので見てもらいたい。

郵便番号→住所検索Ajax by シェルスクリプト デモ

Perl or PHP or Ruby? MySQL or PostgreSQL? prototype.js or jQuery?

これくらいのことで、そんなの要らないよ。

もっと、素のUNIXの力を活かそうぜ!



郵便番号

住所検索
(自前の辞書で)

住所(都道府県名)

住所(市区町村名)

住所(町名)

[解説+ソースコードをしてみる](#)

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/

あとかき

● 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）といえます。

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ください。



● 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ださい」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 改訂版 — POSIX 原理主義を貫く

2014 年 12 月 30 日 初版発行
2015 年 3 月 28 日 第二版発行
2015 年 x 月 x 日 第三版発行

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 株式会社イニユニック
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