いろんな音を聴いて、世界が広がる

USP MAGAZINE

for the sophisticated shell scripters

日本で唯一のシェルスクリプト総合誌

よりぬき版

2014 6 June Vol. 14

¥500

プロフェッショナル達の素顔
神田京子さんに訊く、真打の話芸

新連載
法林浩之の FIGHTING TALKS

人間とコンピュータの可能性

ユニケージエンジニアの作法

USP MAGAZINE

2014 June vol.14

Contents

第3回

Haskellでやってはいかんのか?

written by
産業技術大学院
・USP 研究所・USP 友の会[1]
上田隆一

USP友の会のシェル芸勉強会[2] は、日々、他の言語からの他流試合に晒されているのである[3]。そこで上田は、Haskellで自ら他流試合を行い、さらにシェル芸勉強会をいじめる自傷行為に手を染めるのであった。

1 はじめに

皆さん、酒飲んだくれてますか？ 富山の生んだブラックエンジェル[4] 上田です。

2 今回の問題：シェル芸勉強会第１回２問目

……と、挨拶の後、別に面白いことも思い浮かばなかったのでさっさと本題に行きます。 今日からは第３回にしてやっと２問目です。[5]

/etc/passwd から、次を調べてください。
「ログインシェルが bash のユーザーと sh のユーザー、どっちが多い？」

たぶん私がこんな問題を出されたら、解答はこうです。

知るかボケ。……すんません（大学受験で出題された 400 字作文でこんな解答をして 浪人の遠因となった過去が）。真面目にやります。やりません。いや、やりますやります。

事前準備として、次のように cabal というツールで split というパッケージをインストールしておきます。root 権限でやりましょう。

リスト１

```
1 $ sudo cabal install split
```

cabal がインストールされていない場合は、第１回でもやりましたが、次のようにインストールします。

リスト２：ubuntu の場合

```
1 ueda@ubuntu:~$ sudo apt-get install haskell-platform
```

リスト３：Mac の場合

```
1 uedamac:~ ueda$ brew install haskell-platform
```

また、入力する /etc/passwd は Linux のものを想定しています。[6]

3 手始め：/etc/password の最後のフィールドを取得

1. 順に助教、アドバイザリーフェロー、会長
2. シェルのワンライナー勉強会
3. Ruby、Perl、PowerShell 等々。
4. 略すと富山ブラック
5. 遅い。

まず /etc/passwd から、集計に必要な部分だけ取得することにしましょう。シェル芸ならリスト４のような感じですね。

リスト４：シェル芸でシェルのリストを作る

```
1 $ awk -F: '{print $NF}' /etc/passwd | head
2 /bin/bash
3 /bin/sh
4 /bin/sh
5 /bin/sh
6 ...
```

さて、対応する Haskell のコードはリスト５のようになります。

リスト５：q1_2_1.hs

```
1 $ cat q1_2_1.hs
2 import Data.List.Split
3
4 main = getContents >>= putStr . main'
5
6 main' :: String -> String
7 main' cs = unlines $ map getShell (lines cs)
8
9 getShell :: String -> String
10 getShell ln = last $ splitOn ":" ln
```

関数は三つです。４行目の main 関数は前回の問題でも出てきましたが、標準入力から字を読み込んで標準出力に加工した字を出す関数です。

6、７行目の関数の説明はすっ飛ばして先に 9、10 行目をやっつけましょう。この関数は、/etc/passwd の一行を入力に受け付けます。受け付ける文字列は、例えば

リスト６

```
1 ueda:x:1000:1000:Ryuichi Ueda,,,:/home/ueda:/bin/bash
```

のようなもので、この関数は、コロン区切りの最後のデータ /bin/bash を返します。ちょっと ghci でやってみましょう。リスト７のようになります。

リスト７：関数が無いと叱られる

```
1 $ ghci
2 GHCi, version 7.4.1: http://www.haskell.org/ghc
  /  :? for help
3 (略)
4 Prelude> let getShell ln = last $ splitOn ":" l
  n
5
6 <interactive>:2:26:
7     Not in scope: `splitOn'
8     Perhaps you meant `splitAt' (imported from
  Prelude)
```

……なんか怒られてしまいました。 ちゃんと読むと[7]「splitOn なんて関数ねえぞ！」と言っています。

これを解消する鍵はリスト 5、q1_2_1.hs のコードの１行目にあります。「import Data.List.Split」とありますが、これは Data.List.Split という「モジュール」をインポート、つまり使うという意味になります。先ほど cabal でインストールしたのがこのモジュールに対応するパッケージです。中身は関数だらけです。

さて、これを知った上でもう一度 ghci でリスト８のように試してみましょう。

リスト８：適切なモジュールを import すると怒られない

```
1 Prelude> import Data.List.Split
2 Prelude Data.List.Split> let getShell ln = last
    $ splitOn ":" ln
3 Loading package split-0.2.2 ... linking ... don
  e.
4 Prelude Data.List.Split> getShell "aaa:bbb:ccc"
5 "ccc"
```

うまくいきました。

getShell の中身について説明すると、splitOn が、指定した文字列で文字列を切ってリスト化する関数です。

リスト９

```
1 Prelude Data.List.Split> splitOn "aho" "thisaho
  thataho"
2 ["this","that",""]
```

last は、リストの最後の要素を返します。

リスト 10

```
1 Prelude Data.List.Split> last [1,2,3]
```

6. Mac でやってもよいのですが、なーんか /etc/passwd がぐちゃぐちゃでよく分からないんですよね……。
7. ちゃんと読みましょう。

```
2 3
```

では、6、7行目の説明を。6、7行目は、標準入力から読み込まれた字を行ごとに分割してリストにして（lines csの部分）、mapで各行にgetShellを適用して、mapの返した文字列のリストをunlinesで改行を入れてリストをくっつけて返しています。

これもghciで試せばよいですね。リスト11のようにやってみましょう。便利ですね〜。ghciで改行を指定するときは、¥nと書いておけば大丈夫です。

リスト11：ghciでmain´を試す

```
1 Prelude Data.List.Split> let cs = "aa:bb¥ncc:dd
  "
2 Prelude Data.List.Split> lines cs
3 ["aa:bb","cc:dd"]
4 Prelude Data.List.Split> map getShell (lines cs
  )
5 ["bb","dd"]
6 Prelude Data.List.Split> unlines $ map getShell
   (lines cs)
7 "bb¥ndd¥n"
```

このような、関数を組み合わせるプロセスはシェル芸にも通じるところがあります。入出力だけ考えるという点で、関数型言語とシェル芸は通じるところがあります。

q1_2_1.hsをコンパイルしてリスト12のように出力を見ておきましょう。

リスト12：q1_2_1.hsの使用

```
1 ueda@remote:~/GIT/USPMAG/SRC$ cat /etc/passwd |
   ./q1_2_1
2 /bin/bash
3 /bin/sh
4 /bin/sh
5 /bin/sh
6 /bin/sync
7 /bin/sh
8 ...
```

4 さて、数えましょう

1. 関数を新設してundefinedで型チェック

では、今度は数える部分を書きましょう。先ほどのq1_2_1.hsはmap getShell（lines cs）の出力をunlinesしてそのまま出力していましたが、このunlinesに入力する前にシェルの種類を数える関数を挟めばよいことになります。まず、この考えをそのままコードにします。リスト13のように書きました。

リスト13：q1_2_2.hs

```
1 $ cat q1_2_2.hs
2 import Data.List.Split
3
4 main = getContents >>= putStr . main´
5
6 main´ :: String -> String
7 main´ cs = unlines $ shellCount $ map getShell
  (lines cs)
8
9 getShell :: String -> String
10 getShell ln = last $ splitOn ":" ln
11
12 shellCount :: [String] -> [String]
13 shellCount = undefined
```

7行目のunlinesの後ろ[8]にshellCountという関数をはさんで、shellCountという関数を12、13行目で定義しています。……といっても、13行目にundefinedとあるように、まだ定義していません。

何でこんなことをするかというと、コンパイラにかけるためで、undefinedと書いておくと

リスト14

```
1 ueda@remote:~/GIT/USPMAG/SRC$ ghc q1_2_2.hs
2 [1 of 1] Compiling Main             ( q1_2_2.hs
  , q1_2_2.o )
3 Linking q1_2_2 ...
```

というように通ります。当然実行時にエラーが出ますが。ただ、こうやってコンパイラに通すと型をチェックすることができます。そのため、わざわざundefinedと書いてコンパイルしてみたのでした。

2. shellCountを実装（前半）

では、shellCountを書いたものを示します。と言いたいところなのですが、思ってたよりややこしかったので、途中までのコードをリスト15に示します。

8. 処理的には手前

リスト 15：q1_2_3.hs

```
 1 ueda@remote:~/GIT/USPMAG/SRC$ cat q1_2_3.hs
 2 import Data.List
 3 import Data.List.Split
 4
 5 main = getContents >>= putStr . main'
 6
 7 main' :: String -> String
 8 main' cs = unlines $ shellCount $ map getShell (lines cs)
 9
10 getShell :: String -> String
11 getShell ln = last $ splitOn ":" ln
12
13 shellCount :: [String] -> [String]
14 shellCount shs = map f $ shellCount' [ (1,s) | s <- (sort shs) ]
15     where f (n,str) = unwords [show n,str]
16
17 shellCount' :: [(Int,String)] -> [(Int,String)]
18 shellCount' shs = shs
```

このコードをコンパイルして実行するとリスト 16 のようになります。

リスト 16：q1_2_3.hs のコンパイル

```
 1 ueda@remote:~/GIT/USPMAG/SRC$ ghc q1_2_3.hs
 2 [1 of 1] Compiling Main             ( q1_2_3.hs
   , q1_2_3.o )
 3 Linking q1_2_3 ...
 4 ueda@remote:~/GIT/USPMAG/SRC$ cat /etc/passwd |
   ./q1_2_3
 5 1 /bin/bash
 6 1 /bin/bash
 7 1 /bin/false
 8 1 /bin/false
 9 1 /bin/false
10 1 /bin/false
11 1 /bin/sh
12 ...
```

つまり、シェルのパスに個数をくっつけたデータになりました。シェルの名前はソート済みですので、あとは例えば Open usp Tukubai を使ってリスト 17 のようにやれば答えが出てきてしまうのですが[9]、もうちょっと Haskell で辛抱しましょう……。

リスト 17：q1_2_3 から Tukubai を使って答えを出す

```
1 ueda@remote:~/GIT/USPMAG/SRC$ cat /etc/passwd |
  ./q1_2_3 | self 2 1 | sm2 1 1 2 2
2 /bin/bash 2
3 /bin/false 4
4 /bin/sh 17
5 /bin/sync 1
6 /usr/sbin/nologin 1
```

さて、q1_2_3.hs のコードを読んでいきます。まず、17、18 行目の shellCount' は、今のところ入力をそのまま出力しています。この関数の実装は後回しにします。同じ型のものを返す関数を仮に書くときは undefined よりもこのように同じものを返しておいた方がデバッグに便利です。17 行目にあるように、この関数の型は

```
shellCount' :: [(Int,String)] -> [(Int,String)]
```

です。(a,b)（ここでは a,b はそれぞれ Int と String）というものがありますが、これは「タプル」というもので、二つの型の値を組み合わせるときに使います。ですのでこの関数の型は「Int と String のタプルのリストを入出力する」ということになります。

んで、今回作ったメインのものは 13 ～ 15 行目の shellCount です。ここには新しい書き方が二つも！

まず、15 行目の where ですが、where と書くと、関数の中で関数を定義できます。ここで定義しているのは関数 f で、こいつは何をやっているかというと、

9. ああ……答えを書いてしまった……。

数字 (Int) と文字列 (String) のタプルを入力にもらって、数字を文字列化し、文字列とくっつけて出力しています。ghci で等価なコードを試してみましょう。

リスト 18

```
1 Prelude> [show 5, "abcde"]  <- 5 がf の引数のn, "abcde" がstr に相当
2 ["5", "abcde"]
3 Prelude> unwords [show 5, "abcde"]
4 "5 abcde"
```

次に [と] で囲まれた部分ですが、これはリスト内包というものです。これもクドクド説明するより例で示した方がいいですね。例えば下の例だと、リスト [1, 2, 3] から一つずつ要素を x に結びつけて、それに一つずつ 2 を足してリストにします。

リスト 19

```
1 Prelude> [ x+2 | x <- [1, 2, 3] ]
2 [3, 4, 5]
```

実は、次の map を使った例と等価です。

リスト 20

```
1 Prelude> map (+2) [1, 2, 3]
2 [3, 4, 5]
```

んで、q1_2_3.hs のコードでは、タプルを作るためにリスト内包を使っています。具体例を下に示します。例の中で sort も使ってみます。

リスト 21

```
1 Prelude> import Data.List      <- sort を使うためにimport
2 Prelude Data.List> [ (1, s) | s <- (sort ["bbb", "aaa", "ccc"] ) ]
3 [(1, "aaa"), (1, "bbb"), (1, "ccc")]
```

さあ、最後に shellCount′ を実装します。今のところタプルの数字、つまり個数はすべて「1」で出力していますが、これを同じシェルでまとめていきます。

さて次を……というところですが、紙面がもうございません。

今月はこれでお開きにしたいと思います。

5 おわりに

今回はシェル芸勉強会第 1 回の 2 問目を扱いました。本編ではあまり言及しませんでしたが、今回は問題を関数に分けていくプロセスが随所で見られました。Haskell のコードを書くときは、このように頭を整理してどこで関数を分けることができるのか、探りながら書いていくことになります。実はこれ、どんな言語でも通用する発想方法ですので、何の言語を使うにせよ、よいトレーニングになるかと。つまりは Haskell やれということで、今回を締めさせていただきます。

次回は続きからということで、まーた途中になっちまいましたが、来月まで辛抱強くお待ちいただければ……と。

● お知らせ：コード募集

本稿で出た問題の Haskell のコードを、名前あるいはハンドルネームと共に送ってください。 短いもの、あるいは変態的なものをお願いいたします。

email: mag@usp-lab.com

教えて先輩♡

サーバー運用お助けTips

(っ´∀`)っ

第2回

written by 濱田 康貴

みなさまこんにちは。(っ´∀`)っ ゃー こと@nullpopopoでございます。
内地の梅雨は毎年のことながらゲンナリするものですが、この時期の札幌は暑くもなく寒くもなく風が爽やかな季節です。昼休みにはよく1本道を自転車でスイスイ走り、
午後の仕事に向けて気分をリフレッシュしたことを思い出します。
そんな1本道のように一気通貫一撃必殺で仕事をこなすコマンドラインの技がありまして、
人はこれを「ワンライナー（一行野郎）」と呼びます。
今回はワンライナーを駆使して仕事を素早く楽にこなす術をご紹介しようと思います。

" ワンライナーとは、ある処理を行うためにプログラムを書かず（書くまでもなく）、コマンドラインを 1 行に並べて実行する。"

これだけでは何のことやら？と思われるでしょう。最近では、ワンライナーよりも「シェル芸」という言葉に馴染みを持たれている方もいらっしゃるかと思いますが、シェル芸の創始者、USP 友の会会長 上田さんの言葉を借りてみましょう。

" シェル芸とはマウスも使わず、プログラムも書かず、GUI ツールを立ち上げる間もなく、あらゆる調査・計算・ファイル処理をコマンド入力一撃で終わらす。"

ワンライナーでできること

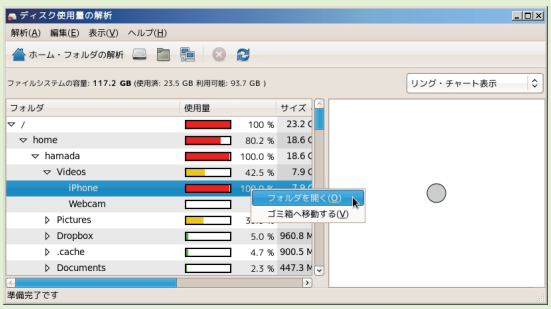
ワンライナーやシェル芸の説明で、どんなことができるのかが少しイメージしやすくなったかと思います。コマンドラインに苦手意識を持たれている方は、「マウスでぽちぽちクリックしたほうが早いじゃん」と思われるかも知れませんが、例えば、「ある Linux サーバーで一番容量を消費しているファイルがあるディレクトリを探し、容量の大きいファイルのトップ 10 を見せてくれ」と上司から依頼されたとしましょう。GUI でぽちぽちクリックする方法とワンライナーで、同じ目的の作業を比較してみます。ここでは、CentOS 6.5 のサーバー環境を想定しています。

◆ 1. GUI でぽちぽち編

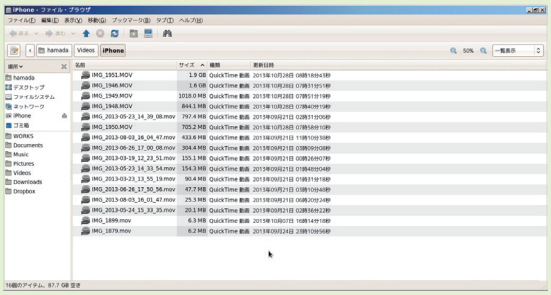
Gnome のメニューから「アプリケーション」を開き、「システムツール」から「ディスク使用量の解析」を起動。「ファイルシステムの解析」ボタンをクリックしてから、使用量のゲージが多い順にクリックする。

その後、フォルダを開いて容量の順に表示をソートしてスクリーンショットを撮るかファイル名をテキストにコピーしてメールで上司に報告する。

【図 1】「ディスク使用量の解析」画面



【図 2】フォルダにソートして表示する



※ 前提条件として「gnome-utils」パッケージがインストールされていること

◆ 2. ワンライナー編

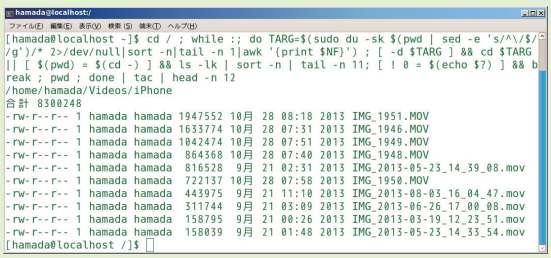
ターミナルを開き、リスト 1 のワンライナーを実行する。

▶リスト 1

```
[root@localhost ~]# cd / ; while :; do TARG=$(du -
sk $(pwd | sed -e 's/^¥/$//g')/* 2>/dev/null|sort
-n|tail -n 1|awk '{print $NF}') ; [ -d $TARG ] &&
cd $TARG || [ $(pwd) = $(cd -) ] && ls -lk | sort
-n | tail -n 11; [ ! 0 = $(echo $?) ] && break ; p
wd ; done | tac | head -n 12
```

これを実行するだけ（USP MAGAZINE Vol.12 拙著「教えて先輩♡ サーバー運用お助け Tips」より引用および改変）。実行結果例は図 3 参照。詳細は " おまけの解説 " で説明いたします。

【図 3】



```
[hamada@localhost ~]$ cd / ; while :; do TARG=$(sudo du -sk $(pwd | sed -e 's/^\/$/
/g')/* 2>/dev/null|sort -n|tail -n 1|awk '{print $NF}') ; [ -d $TARG ] && cd $TARG
|| [ $(pwd) = $(cd -) ] && ls -lk | sort -n | tail -n 11; [ ! 0 = $(echo $?) ] && b
reak ; pwd ; done | tac | head -n 12
/home/hamada/Videos/iPhone
合計 8300248
-rw-r--r-- 1 hamada hamada 1947552 10月 28 08:18 2013 IMG_1951.MOV
-rw-r--r-- 1 hamada hamada 1653774 10月 28 07:31 2013 IMG_1946.MOV
-rw-r--r-- 1 hamada hamada 1042474 10月 28 07:51 2013 IMG_1949.MOV
-rw-r--r-- 1 hamada hamada  864368 10月 28 07:40 2013 IMG_1948.MOV
-rw-r--r-- 1 hamada hamada  816528  9月 21 02:31 2013 IMG_2013-05-23_14_39_08.mov
-rw-r--r-- 1 hamada hamada  722137 10月 28 07:58 2013 IMG_1950.MOV
-rw-r--r-- 1 hamada hamada  443975  9月 21 11:10 2013 IMG_2013-08-03_16_04_47.mov
-rw-r--r-- 1 hamada hamada  311744  9月 21 03:09 2013 IMG_2013-06-26_17_00_08.mov
-rw-r--r-- 1 hamada hamada  158795  9月 21 00:26 2013 IMG_2013-03-19_12_23_51.mov
-rw-r--r-- 1 hamada hamada  158039  9月 21 01:48 2013 IMG_2013-05-23_14_33_54.mov
[hamada@localhost /]$
```

※ 前提条件として「gawk」「coreutils」「sed」パッケージがインストールされていること

GUI でぽちぽちの方が楽で、しかも前提条件が緩そうですが、そもそも「gawk」「coreutils」「sed」パッケージは GUI の有無に関係なくどんなマシンにもインストールされています。仮に、サーバーマシンで GUI を使う目的で「gnome-utils」をインストールするには少なくとも「avahi-glib」「dmz-cursor-themes」「flac」「gnome-icon-theme」「gnome-panel-libs」「gnome-themes」「gnome-utils-libs」「gnome-vfs2」「gstreamer」「gstreamer-tools」「gtk2-engines」「libasyncns」「libbonobo」「libbonoboui」「libcanberra」「libcanberra-gtk2」「libglade2」「libgnome」「libgnomecanvas」「libgtop2」「libogg」「libsndfile」「libtdb」「libvorbis」「pulseaudio-libs」「rarian」「rarian-compat」「sound-theme-freedesktop」「system-gnome-theme」「system-icon-theme」「xml-common」がインストールされていないとなりません。

自宅の PC ならまだしも、サーバーマシンにこれだけ多くのパッケージをインストールするということは、yum などのパッケージ管理を使うとしてもアップデートの妥当性を考えなければなりません。そうすると、1 つのサーバーマシンにインストールするパッケージの数は少なければ少ないほど運用が楽になります。

さらに、毎日同じ作業を繰り返すとなると、GUI 操作を行うために人が張り付かなければなりません。一方、ワンライナーであれば、テキストのコピペで再実行が容易で、かつ何度繰り返しても同じ条件のアウトプットができます。自動化だって、cron から起動してメールで飛ばすようにすればそれでおしまいです。他人に同じ仕事をしてもらうのだって、GUI を人にクリックしてもらうにはテキストの説明だけでは何かと不安です。さりとて図解入りの手順書を作る作業に時間をかけたくないですよね？

しかし、ワンライナーであれば「これコピペして実行して結果をメールにコピペしてちょうだい」と伝えるだけです。後ろ向きな仕事やそれに伴うコミュニケーションに時間をかけるくらいなら、仕事はさっさと終わらせてコーヒーでも飲みながら同僚と雑談していたほうがよっぽど職場によい空気をもたらすと思いませんか？

ワンライナーの書き方

コマンドを 1 行で繋げて実行するのがワンライナーというのはこれまでの例で理解できたかと思います。ではどうやって 1 行にするのか？についてもう少しパターンを整理したいと思います。

（1）複数のコマンドをとにかく 1 行で実行する

例えば、サーバーのロードアベレージとメモリ使用量、ディスク使用率を見たい場合、単純にコマンドを

「;」で区切って１行で実行することができます。

▶リスト２

```
[hamada@localhost ~]$ w ; free -m ; df -h ↵
 13:19:50 up 1 day, 18:21,  2 users,  load average
: 0.07, 0.06, 0.02
USER     TTY      FROM              LOGIN@   IDLE
  JCPU   PCPU WHAT
hamada   tty1     :0                Thu18   42:21m
30:24   0.02s pam: gdm-password
hamada   pts/0    :0.0              13:11    0.00s
 0.06s  0.00s w
             total       used       free     share
d    buffers     cached
Mem:          7779       3254       4525
0         192       1198
-/+ buffers/cache:       1862       5916
Swap:          199          0        199
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda3       117G   41G   71G  37% /
tmpfs           3.8G  9.6M  3.8G   1% /dev/shm
/dev/sda1       194M  122M   63M  67% /boot
```

しかし、これだけでは見づらいので、各項目の表題をつけたり空行をつけてみたくなるでしょう。次の「(2) コマンドの出力をパイプで区切る」で、もう少し見やすく整形してみることにします。

(2) コマンドの出力をパイプで区切る

コマンドの出力をパイプで区切ることにより、前に実行したコマンドの出力結果を次のコマンドに食わせることができます。

▶リスト３

```
[hamada@localhost ~]$ echo "ロードアベレージ" ; w
| grep "load average" | awk ´{print $8,$9,$10,$11,
$12}´ ; echo ; echo "メモリ使用量" ; free -m ; ech
o ; echo "ディスク使用率" ; df -h ↵
ロードアベレージ
load average: 0.02, 0.03, 0.00

メモリ使用量
             total       used       free     share
d    buffers     cached
Mem:          7779       3239       4539
0         192       1197
-/+ buffers/cache:       1849       5930
Swap:          199          0        199

ディスク使用率
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda3       117G   41G   71G  37% /
tmpfs           3.8G  9.6M  3.8G   1% /dev/shm
/dev/sda1       194M  122M   63M  67% /boot
```

w コマンドはログインしている人やその人がやっていることを表示するコマンドですが、ヘッダには、現在の時刻、システムが稼働している期間、現在ログインしているユーザーの数、過去 1、5、15 分でのシステムの平均負荷が順に表示されるので、ロードアベレージだけを表示させるよう grep と awk で整形しています（※ 1）。grep で「load average」が表示されている行を捕まえ、awk コマンドで８～ 12 列目を抜き出しています。また、すべてのコマンドを実行する前に、echo コマンドでタイトルをつけたり、echo の後に引数を与えずコマンド実行後に空行をつけたりしています。

こうしてパイプで区切れるということは、１回１回サーバーにログインせずとも毎日決まった時間にメールでコマンド出力結果を受け取れるということです。しかし、複数コマンドをまとめて１通のメールとして送信するには少し工夫が必要です。

▶リスト４

```
[hamada@localhost ~]$ for A in :; do echo "ロード
アベレージ"; w | grep "load average" | awk ´{print
 $8,$9,$10,$11,$12}´; echo ; echo "メモリ使用量";
free -m ; echo; echo "ディスク使用率" ; df -h ; do
ne | mail -s "$(date) $(uname -n) SYSTEM STATUS" h
amada@localhost
```

先ほどのコマンドを、bash の for 文で囲って１つの塊にし、これをパイプラインで mail コマンドに渡してあげています。for 文は「for A in 1 2 3; do echo $A; done」などのように、愚直に変数 A に代入した値のぶんだけ実行する使い方が一般的かと思いますが、ここではちょっとだけトリッキーに「: (true)」コマンドを代入しています。こうすることで、for 文の do から done までに囲まれたコマンドを１回だけループ実行することができます。あとはこのワンライナーを cron に登録してあげれば、わざわざターミナルにログインしなくてもサーバーの状態を簡単に見ることができるでしょう。

(3) 前に実行したコマンドの成功と失敗によって処理を使い分ける

ここまでのワンライナーでは、コマンドの区切りが２つ登場しました。「: (true コマンド)」と「| (パイプライン)」です。しかし、条件によって処理を使い分けたいこともあるでしょう。次の例では、ロードアベレージが１を超えたときとそうでないときでメッセー

ジを変えたい場合の使い分けを取り上げてみましょう。直近のコマンドが正常終了であれば、「&&」の後に続くコマンドを実行します。直近のコマンドが異常終了であれば、「||」の後に続くコマンドを実行します。

▶リスト5

```
[hamada@localhost ~]$ Threshold=4 ; NUM=$(echo "$(w | grep "load average" | awk ´{print $10}´ | sed -e s/,//g)*100" | bc) ; [ $(expr ${Threshold} ¥* 100) -gt $(echo $NUM | awk ´BEGIN {FS="."} {print $1}´) ] && echo "ロードアベレージは${Threshold}より小さいので正常域です" || echo "ロードアベレージは${Threshold}を超えました"
```

サーバーのロードアベレージが、ある閾値を超えたかどうかによって出力するメッセージを変えてみました。今回の例では閾値を「Threshold」という変数で指定します。次に、直近のロードアベレージを w コマンドから抜き出すのですが、test コマンドの中で閾値より小さいかどうかの比較をするのに小数点の計算ができないので、「NUM」変数で 100 をかけてあげます。同じく、test コマンドの中で「Threshold」変数の値も 100 倍にしてあげています。そして、ロードアベレージが閾値より小さければ正常終了なので、このワンライナーを実行した後は「ロードアベレージは 4 より小さいので正常域です」と表示されます。何らかの理由で CPU 処理の待ちが多いと、「ロードアベレージは 4 を超えました」と表示されます。

test コマンドのオプションは数値の比較のほか、ファイルの有無や文字列比較も行うことができますので、是非使いこなしてみてください。あまり条件を複雑にしすぎても大変かと思いますが、エラーハンドリングをきちんと考えてワンライナーを実行したりプログラムを書いたりするクセをつけておけば、何かエラーがおきたときの対処を早めることに繋がります。

ワンライナーの書き方まとめ

それでは、ここまでのまとめを見てみましょう。

- **コマンドの単純な区切りは「;」で行う**
  command1 ; command2 ; ...
- **コマンド出力結果を次のコマンドに食わせる**
  command1 | command2 | ...
- **test コマンドを使い、条件分岐を行う**
  [条件 1 オプション 条件 2] && test コマンドが正常終了したときに実行したいコマンド || test コマンドが異常終了したときに実行したいコマンド

1 つ補足しますと、「&&」や「||」の前は必ず test コマンドでなければいけない、というわけではなく、直前に実行したコマンドが正常終了したかどうかで処理を分けたい場合に使うと便利です。例えばこういう使い方もできます。

▶リスト6

```
[hamada@localhost ~]$ touch hoge && echo "hogeファイルを生成できました" || echo "hogeファイルの生成に失敗しました。ディレクトリのパーミッションを確認してください"
```

おまけの解説

ワンライナー編のリスト 1 を詳しく解説します。

/ ディレクトリから順に、du コマンドで一番容量を消費しているディレクトリへ cd していきます。なので最初に「cd /」しています。while のループの中身ですが、du コマンドでカレントディレクトリ直下で一番容量を食っているファイルやディレクトリを「TARG」変数に入れています。そして、これがディレクトリだったら cd して、ループの先頭に戻ります。最終的に変数 TARG がファイルになるまでこれを繰り返し、今いるディレクトリと直前にいたディレクトリが同じになったところで ls の結果をキロバイト単位でソートして表示し、現在のディレクトリを表示してループを抜けます。あとは逆順にならべて head で整形しておしまいです。

いかがでしたでしょうか。ワンライナーを使うメリット、そして何ができるのかがイメージできると、明日からの仕事がよりよいものになるでしょう。

※1　jman より引用
http://linuxjm.sourceforge.jp/html/procps/man1/w.1.html

未来に活きる！ 現場で使える！

データモデリング

第４回　イベント系のモデリング

written by **熊野憲辰**

第3回より具体的なデータモデリングについて解説している。前回は「マスタ系」のモデルを考察した。今回は「イベント系」のモデルについて述べる。イベントとは、企業活動における出来事そのものを指し示している。

順序と多重度

モデリングの観点から見た時、企業活動の中における「出来事」とは何を表わす言葉だろうか。例を挙げると、受注した、仕入れた、出荷した、入荷した、などがここでいう「出来事」に該当する。出来事の発生は「伝票」という体裁で認識することができる。また前回触れたように、イベントには必ず日付が属性として帰属する。

ここで一般的な伝票というものをイメージしてほしい。伝票のヘッダーには日付や取引先などの情報が、明細には商品や数量、金額が記載されている。つまり「伝票における明細」こそが、最も粒度の小さいイベントを表わしていることを理解できる。

この伝票明細を素直に記録していくことがイベント系のモデルにおいて重要になる。

ご存知のようにデータベースは、集計処理が大得意である。最も細かな明細単位で、企業活動の出来事を漏れなく記録していけば、それを集計したり加工したりという処理は単純なアルゴリズムで実現できる。伝票明細レベルでイベントを漏れなく素直に記録できていれば、後はなんでもござれ、というわけだ。

ただし、この「出来事を漏れなく素直に記録する」という作業には、ややコツが要る。出来事には「先行」と「後続」があるからだ。ある出来事が発生した後に、次の出来事が発生する、またはそうでなくてはならない、といった「順序」がたいへん重要である。受注という出来事の後に、出荷という出来事が起きるというのは、まさにそうした順序の一例だ。

その上で留意したい点は、出来事の先行・後続における「多重度」がモデリングを難しくする一因にあること。受注〜出荷という先行・後続イベントを例に、いくつかのモデル事例を解説していこう。

●パターン 1

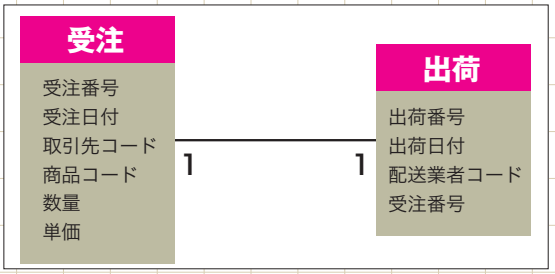
先行・後続イベントが 1:1 で起きるケース

受注 1 件に対して、出荷というイベントが 1 対 1 で発生するパターンになる（図 1）。

受注に関する詳細な属性は、受注エンティティに保持されている。出荷エンティティ側には、「出荷番号」「出荷日付」「配送業者コード」が主となる属性がある。このモデルのポイントは、出荷エンティティに「受注番号」を保持している、ということだ。すなわち受注番号が、「どの取引先からの受注なのか」という情報、または商品、数量、単価などの属性を代表している。しかし、こうなっていないモデルをよく見かける。端的なのは出荷エンティティ側に取引先コードや商品コード、数量、単価などを引きずっているケースだ。これは冗長であり正規化されていないモデルといえる。

受注エンティティに帰属する属性、出荷エンティティに帰属する属性を正確に区別してモデル化するようにしよう。

図 1：多重度 1:1 のモデル



●パターン 2

複数の受注が 1 回の出荷となる（N:1）ケース

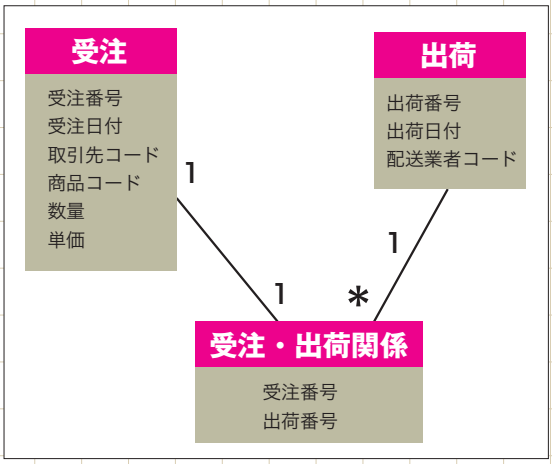
実務では、複数の受注をまとめてから出荷を行うこ

ちらのパターン２（N:1）のほうが、圧倒的に多いだろう。受注１件につき毎回出荷するのは効率が悪いからだ。

この場合、モデル図は、図２となる。受注と出荷の関係は、N:1（図でＮ側は＊で記述）だから、パターン１のように出荷エンティティ側に受注番号を保持させることはできない（複数の受注番号があるため）。だから、受注と出荷のエンティティ間に双方の対応関係を表にして保持する、ということを表わしている。

この対応表で、受注からみた後続の出荷、出荷からみた先行の受注をたどることができる。

図２：多重度 N:1 のモデル



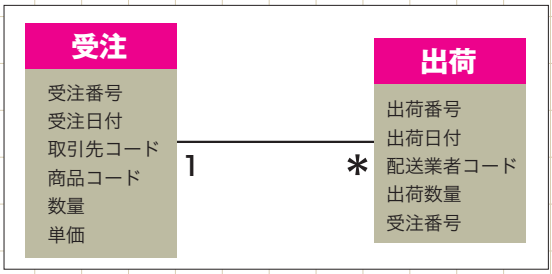
●パターン３

１回の受注が複数の出荷（１：N）となるケース

１回の受注に対して２回、３回に分けて出荷する業務もあるだろう。ある商品を 100 個出荷するという受注に対して、60 個をまず出荷、その後 40 個を別の日に出荷した、というような事例である。

モデル図は、図３となる。出荷エンティティに受注番号を保持しているという点では、図１と同じである。そして、複数の出荷における出荷数量を出荷エンティティに保持している。

図３：多重度 1:N のモデル



受注、出荷ともに、それぞれのエンティティに帰属する属性だけを保持していることを確認してもらいたい。

●パターン４

サブタイプ構造を持つケース

受注と出荷の順序関係において、受注→出荷だけでなく、受注→製造→出荷のようなオーダーメイドのパターンもある。そして、受注エンティティが、サブタイプ構造として、在庫品受注とオーダーメイド受注に分かれるパターンを考える。

在庫品受注は、そのまま出荷に向かう（ここでは、在庫引当、梱包などのイベントは無視している）。オーダーメイド受注は、受注の後、製造イベントを経て出荷となる。これをモデル化すると図４になる（多重度はすべて 1:1）。

図４では、出荷エンティティもサブタイプ構造を持つことになる。そして、在庫品出荷は、先行イベントの番号である受注番号を保持し、オーダーメイド出荷は、製造番号を保持している。

ここで、キーとなる番号の付け方について補足をする。今回のようなモデルで、受注番号の１桁目を、在庫品受注の場合は「1」、オーダーメイド受注の場合は「2」などとやってはいけない。あくまで一つのイベントの番号は単なる連番として設定する。この点についてはマスタ系でもイベント系においても例外はない。

●パターン５

「ヘッダー・明細」構造を含むケース

冒頭で触れたように、企業活動の出来事は、生成された受注伝票や出荷伝票の中のヘッダーと明細によって記述されている。もちろん、在庫引当やピッキングなど、一般的には伝票として記述されないイベントも多々ある。

伝票の体裁は多種多様だが、企業の情報システムで扱われるデータは、ヘッダーがあり、明細行が複数存在する場合がほとんどである。

これまでのパターンで述べてきたような単体のエンティティに、ヘッダーと明細を組み合わせたエンティティが加わると、モデルの示し方に少し難しさが増す。

図５に、先行→後続という関係が、ヘッダー単位で推移する場合のモデルを示そう。

図 4：サブタイプ構造を持つ先行・後続モデル

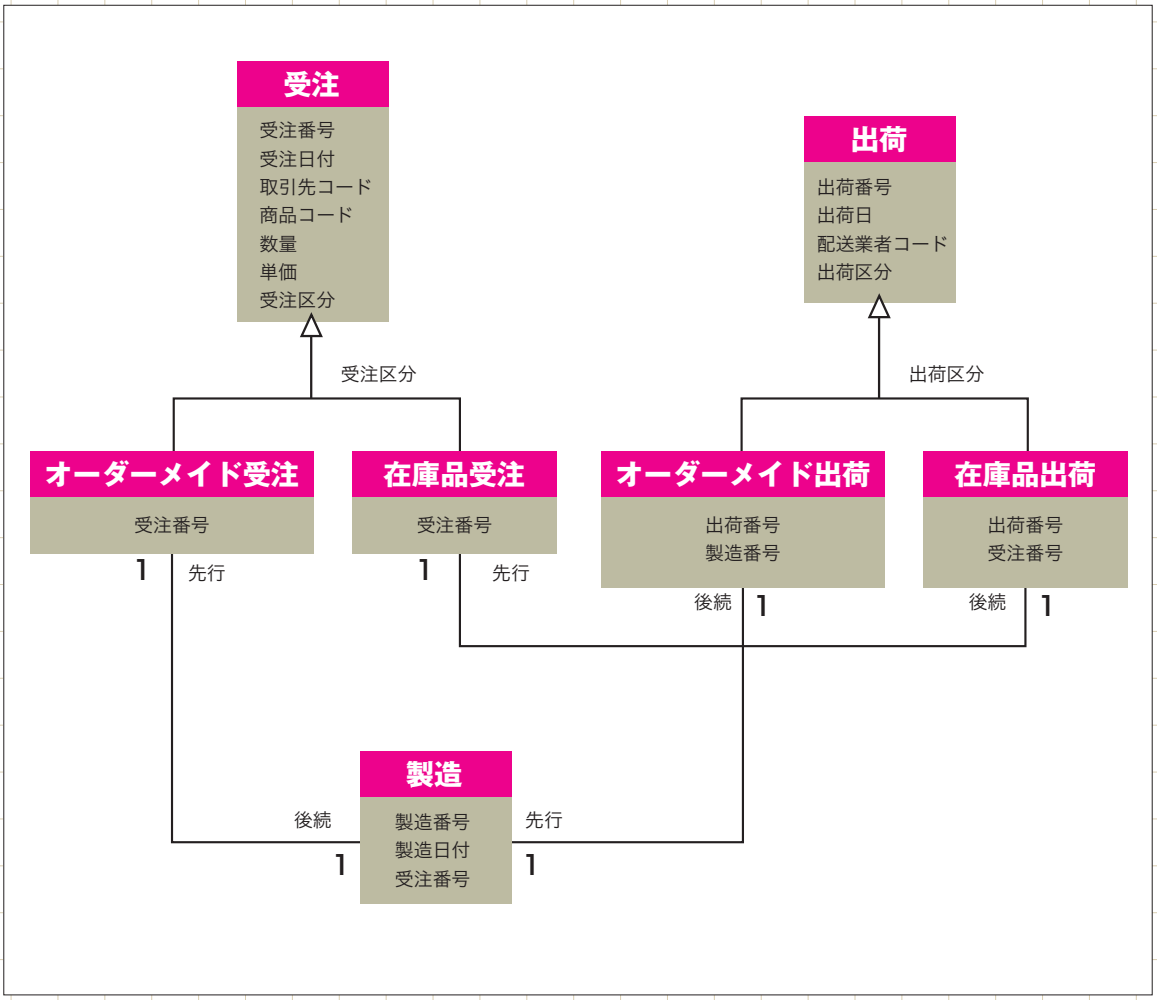


図 5：伝票明細のエンティティを加える

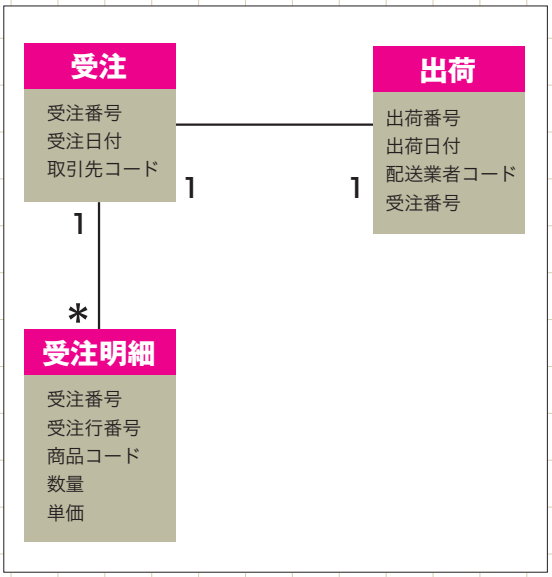
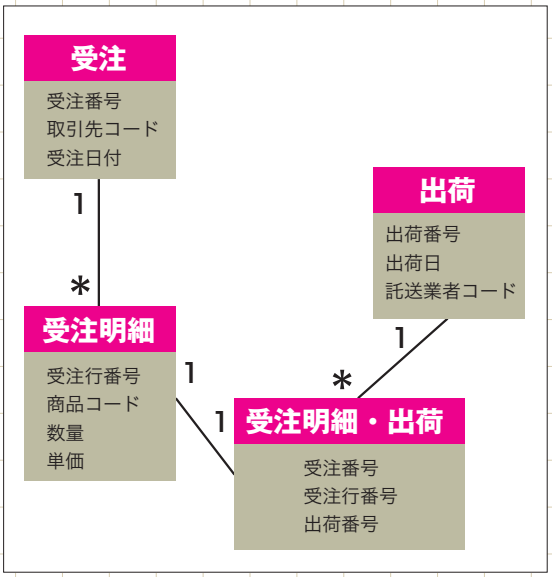


図 6：ヘッダー・明細構造と多重度が追加



●パターン６

明細単位で先行・後続するケース

図６は、ヘッダー・明細構造において先行する受注明細が、後続の出荷につながるパターンのモデル図である。この場合、異なる受注の明細をいくつか集めてから、１回の出荷を行う、ということを表わしており、企業の物流システムにおいて最も一般的なモデルであるといえる。

このモデルでは、複数の受注明細が１回の出荷につながるので、受注明細と出荷のエンティティ間には、パターン２のように、対応関係を記述するテーブルが置かれることになる。このテーブルには、受注番号と受注行番号、出荷番号が保持されている。

イベント系モデルの作成手順

ここまで、先行・後続における各種パターンとみてきた。まとめとして、イベント系のモデルの作成を手順に沿って解説する。

1）業務のトリガーとしてのイベントを見極める

各事例では、受注を業務の先頭に置いているがそうでないことも多いだろう。例えば、見積り。見積りを行った、という事実を記録し、その後続として受注がある、というモデルだ。この場合、見積り内容と実際の受注内容は異なることも予想される。そして、受注にはつながらないことも考えられる。モデル化する時は、これらを考慮する。

見積り作業に関しては、画面からの入力は行うがエンティティとして記録せず、XML ファイルなどでシリアライズして保持するに留める、というのも実践的なアプローチのひとつである。

2）イベントの後続を考慮しサブタイプ構造を洗い出す

前回のマスタ系のモデルについての解説でも述べたが、エンティティをサブタイプ構造で立体化することは、自社業務の整理、可視化に大いに役に立つ。

特にイベント系は、先行・後続の関係が重要なので、これをベースにサブタイプに分類することが有効だ。また、この段階ではヘッダー・明細構造を特に意識せずにスケッチすると、理解しやすい。

3）多重度をモデル化する

イベントの先行・後続が明確になったら、それぞれの多重度を分析してみる。本稿で挙げた事例は、受注からいきなり出荷という流れであったが、実際の物流業務では、在庫引当、ピッキング、梱包などのイベントが途中に存在する。『複数の受注を一括で在庫引当するのかどうか』『ピッキングも、種まき方式なのか、摘み取り方式なのか』などを考慮して多重度を慎重に分析しなければならない。

4）ヘッダー・明細構造の分析

重ねて強調したいが、最初からヘッダー・明細を考慮しないほうがわかりやすい場合が多い。大きな流れや構造を分析した後にヘッダー・明細を与えると混乱が起きにくい。

ただし、伝票や画面、帳票をベースにモデルを作成し、それぞれを統合する場合には、最初からヘッダー・明細の構造を認識することになる。もちろん、このようなボトムアップによるアプローチでモデル化することに問題はない。ただし、イベントの全体像を見失わないように先行・後続、サブタイプ構造などへの目配りは欠かせない。

上記のサービス内容は 2014 年 5 月現在のものです。
最新の情報は弊社ホームページにてご確認ください。

ISBN 978-4-9048-0708-8
C3455 ¥500

9784904807088

USP 研究所
定価（本体 500 円+税）

1923455005002