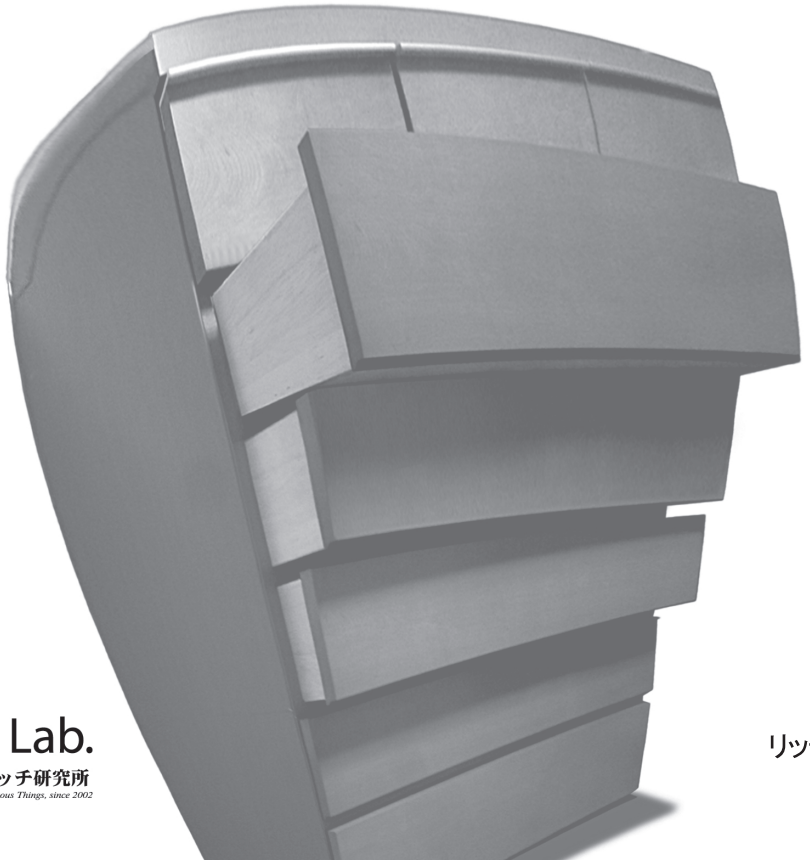
FILE/DIR HACKS

ver. 1.0
SQL to UNIX
マイグレーション編
追加

SQL教徒をPOSIX原理主義者にする
最強データ管理術



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

リッチー大佐 著

File/Dir Hacks

SQL 教徒を POSIX 原理主義者にする
最強データ管理術

リッチー大佐 著

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

まえがき

本書制作の経緯

2014 年頃から 2 年くらいかけて、シェルスクリプト（しかも POSIX の範囲）によるプログラミングを実践するための書籍を作った。Windows/Mac/UNIX のどれでも使えるという非常に高い互換性、しかも一度書いたら 10 年、20 年もの長きにわたり使い続けられるとい持続可能性が他に類のない最大の特長であり、数ある言語の中で「なぜシェルスクリプトを選ぶのか?」という疑問に対する回答を示すことができた。この本はコミックマーケット合わせで何度か改訂版を出したのち、商業出版される本（通称「20 年本」）になり、現在に至っている。

本が多くの人に読まれる中で、様々な要望を頂いた。その中で多かったものは「実用的な例を紹介・解説してほしい」というものだった。確かに 20 年本では、コマンドの使い方や Tips 等、プリミティブな内容に重点が置かれていたように思う。

実用的な例と言われば、我々としても 20 年本で解説し足りないと思っていたことがあった。それが、本書のテーマである「File/Dir Hack」である。20 年本により、高い互換性と持続性を持たせながら UNIX コマンドやシェルスクリプトを使う方法を説明した後は、それらを駆使し、MySQL 等の RDB 製品に頼らずともデータ管理ができることを示したかった。RDB 製品を使えば製品が内部で上手くデータ管理処理をしてくれるからこれだけ普及しているのだと思うが、各製品を取り扱うための専門スキルが要求されるし、シェルスクリプトのような高い互換性・持続性も得られない。ところが、UNIX コマンドやシェルスクリプト、それにファイルやディレクトリーをきちんと駆使すれば、RDB 製品に求めていた多くのことができてしまうのである。

コンピューターにとって、プログラムとデータは二つの大きな柱であり、データを上手く取り扱うことは、プログラムを上手く書くこととはまた別のスキルで、同じくらい大切なものだと考える。20 年本ではカバーしていなかったもう一本の柱を取り上げれば、多くの読者が求めていた、シェルスクリプト・プログラミングの実用的な例という問いに対する回答が示せるのではないだろうか。

ver.1.0（ver.0.2+SQL マイグレーション編）の位置づけ

本書は、ファイル・ディレクトリーの仕組みと、UNIX 系 OS に初めから入っている基本的 UNIX コマンドだけを徹底的に活用することで、RDB 製品やその他ライブラリー類になどに頼ることなく、データ管理を始め、さまざまな処理を一人前にこなそうとするものだ。

0.1 版では手始めに、データをファイルやディレクトリー上にどう格納すべきかを記し、0.2 版ではファイルやディレクトリーを駆使していかにトランザクション処理やタスクキューイングを実現するのかを追記した。

そして 1.0 では、それらの前提も踏まえながら **SQL 文を UNIX コマンド・シェルスクリプトにマイグレーションする方法**を追記した。最終的に書きたかった事はまさにこれであり、（新型コロナウイルス感染拡大による足踏みがありながらも）3 年がかりでようやく完成といえる内容になった。

SQL 文で使用する頻繁に利用するコマンドや句、関数の（全部ではないものの）大半について、UNIX コマンド・シェルスクリプトでどう実装できるのかという辞典として役立つだろう。

ただ、校正が甘いため、誤字等を訂正した修正版（1.01 版）早々に出すかもしれない。

本書の電子版を無償公開

クラウドファンディングで募った資金を元手に、「プロフェッショナル IPv6」という本が発売された。そして同時に、全く同じ内容を収録した電子版も公開されて話題になった。

この活動に敬意を表し（私も購入した）、本書も同じ内容の無償配布をすることにした。是非活用してもらいたい。次の URL から取得できる。

```
https://richlab.org/pub/ebook/fdh1.pdf
```

おことわり

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、下記の宛先へこっそりツッコミなどお寄せ頂きたい。

```
richie.shellshoccar@gmail.com
```

目次

まえがき iii

第 1 章 File/Dir Hack 事始め 1
HACK 1.1 まず、何を重要視するかを確認する 1
1.1.1 File/Dir Hack が目指すもの 1
HACK 1.2 見やすさ・分かりやすさを追求する 2
1.2.1 データはテキストファイルに保管する 2
1.2.2 一発で理解できる見た目を心掛ける 2
HACK 1.3 無理なく速さを追求する 3
HACK 1.4 強靭さ “robustness” を追求する 4
HACK 1.5 データやディスク容量をケチらない 5
1.5.1 ただし、目的無き消費は「無駄」である 6
HACK 1.6 システムを擬人化し、手作業時代の構造を写像する 6
1.6.1 ショッピングカートプログラムが人だったら 7
1.6.2 書類や収納棚のデータ構造をファイル・ディレクトリーに写像する . . . 7
HACK 1.7 File/Dir Hack 用コマンドセット “ShellShoccar” のインストール 8
1.7.1 コマンドセット “ShellShoccar” とは何か 8
1.7.2 コマンドセット “ShellShoccar” をインストール 9

第 2 章 データ管理のためのファイルの使い方 11
HACK 2.1 「列区切りは、半角空白 1 個以上」と決める 11
2.1.1 CSV じゃダメなのか? 11
2.1.2 「半角空白 1 個以上区切り」で、見づらさを解消する 12
2.1.3 「半角空白 1 個以上区切り」は、UNIX コマンドにとっても都合がいい . 14
2.1.4 値として半角空白・タブを含ませたい場合はどうするの? 14
HACK 2.2 文字列中の半角空白は、“_”（アンダースコア）で表す 14
HACK 2.3 改行・タブ等のその他特殊文字はバックスラッシュ記号でエスケープする 15
2.3.1 エスケープ・アンエスケープのコード例 16

HACK 2.4 NULL 値は、“-” や “*” などの記号 1 文字で表す 17
2.4.1 NULL 値に頼るのは「設計の敗北」である 19
HACK 2.5 ファイルフォーマットの種類を把握する 19
HACK 2.6 SSV 形式（広義の field 形式） 20
HACK 2.7 key-value 形式（name 形式） 21
2.7.1 join したくば 2 列固定の SSV 形式 22
HACK 2.8 JSONPath-value 形式・XPath-value 形式 23
2.8.1 基本アイデア 23
2.8.2 変換コマンドと応用例 23
HACK 2.9 CSVindex-value 形式 25
2.9.1 変換コマンドと具体例 25
HACK 2.10 ファイルは、常にソート済の状態での保存を心がける 26
2.10.1 何のためにソートするのか? 27
HACK 2.11 一時ファイルも必要なら使う 29
2.11.1 一時ファイルをより安全に作成・削除するスニペット 30

第 3 章 データ管理のためのディレクトリーの使い方 33
HACK 3.1 RDB のテーブル 1 つ ≠ ファイル 1 つ 33
3.1.1 ランダムアクセスのテーブル、シーケンシャルアクセスのファイル . . . 34
3.1.2 ファイルなら、階層化して格納できる 34
HACK 3.2 1 つのディレクトリー直下のファイル数 ≦10000 35
3.2.1 同一ディレクトリーに大量のファイルを作る実験 35
3.2.2 考察 36
HACK 3.3 内容更新する 1 つのファイルに収める行数 ≦10000 36
3.3.1 大きい 1 つのファイルと細かい大量のファイル 36
3.3.2 1000 万レコードファイルの持ち方実験 37
3.3.3 1 ファイルに持たせるべきレコード数の上限は? 39
3.3.4 会員データはどう管理するか 39

第 4 章 実は排他制御不要な排他処理 41
HACK 4.1 単調増加カウンターは追記リダイレクションで作る 41
4.1.1 ファイルサイズをカウンターとして使う 41
4.1.2 なぜファイルロック不要なのか 42
4.1.3 デメリットと限界 43
HACK 4.2 複数プロセスから同一ファイルの内容を更新しない設計にする 44
HACK 4.3 ファイル更新の最後には mv コマンドを使う 44
4.3.1 良い例と悪い例 45

HACK 4.4　小さなファイルをディレクトリーに並べる 45
4.4.1　在庫数ファイルによる例 46
4.4.2　有効性が低いケースとその対策 46
HACK 4.5　加工元データファイルに加工後データを上書かない 47
4.5.1　ウォーターフォール書き出しが好ましい理由 48
4.5.2　保管コストとの折り合い 48
HACK 4.6　その他、アトミックな処理を活用する 49
4.6.1　アトミックと見なせるファイル操作 49

第 5 章　排他処理（トランザクション処理） 51
HACK 5.1　排他ロックは「ln などの早い者勝ち」ルールで捌く 51
5.1.1　2 種類のロック 51
5.1.2　排他ロックの考え方 52
5.1.3　排他ロック実装のための具体的なルール 53
5.1.4　排他ロックコマンド “pexlock” 53
5.1.5　制約事項 56
HACK 5.2　共有ロックは、ディレクトリーでメンバー管理する 56
5.2.1　共有ロックの考え方 57
5.2.2　共有ロック管理のためのデータ構造はどうするべきか 57
5.2.3　共有ロック管理のための各種ファイル操作 59
5.2.4　共有ロックコマンド “pshlock” 61
HACK 5.3　セマフォ制御は、共有ロックに最大共有数を設けて対応する 64
5.3.1　セマフォとは 64
5.3.2　ファイル・ディレクトリーを駆使して実現する 64
5.3.3　リソーススターベーション問題 66

第 6 章　非同期キューイング – 食券方式な食堂の注文管理 67
HACK 6.1　キューイングは、食券方式の食堂を真似する 67
6.1.1　飲食店の食券はどういう仕組みか 67
6.1.2　食券方式をファイル・ディレクトリーで再現する 69
HACK 6.2　規模や複雑さが増したら、大規模な食堂の仕組みを真似る 73
6.2.1　大規模な食堂はどうなっているか 73
6.2.2　ファイル・ディレクトリーで再現する 74

第 7 章　SQL to UNIX マイグレーション (1) – コマンド 75
HACK 7.1　SELECT コマンド 75
7.1.1　列名でなく列番号で指定するという文化の違いに注意 76

HACK 7.2 INSERT コマンド . . . 76
7.2.1 ソート順を維持しながら追記する . . . 77
7.2.2 ソート順を維持しながら追記する（安定ソートが必要な場合） . . . 79
HACK 7.3 UPDATE コマンド . . . 80
HACK 7.4 MERGE コマンド . . . 80
HACK 7.5 DELETE コマンド . . . 82
HACK 7.6 TRUNCATE コマンド . . . 83
HACK 7.7 DROP コマンド . . . 84
HACK 7.8 CREATE / ALTER コマンド . . . 84

第 8 章 SQL to UNIX マイグレーション (2) – 選択の句 87
HACK 8.1 FROM 句 . . . 87
HACK 8.2 UNION、UNION ALL 句 . . . 88
HACK 8.3 EXCEPT / MINUS 句 . . . 89
HACK 8.4 INTERSECT 句 . . . 90
HACK 8.5 WHERE 句 . . . 91
8.5.1 日時の大小比較はどう翻訳すべきか . . . 92
HACK 8.6 GROUP BY 句 . . . 93
HACK 8.7 HAVING 句 . . . 95
HACK 8.8 DISTINCT 句 . . . 96
HACK 8.9 [NOT] (LIKE / REGEXP / SIMILAR TO) 句 . . . 97
HACK 8.10 [NOT] BETWEEN 句 . . . 98
HACK 8.11 [NOT] IN 句 . . . 99
HACK 8.12 ANY / SOME 句 . . . 100
HACK 8.13 ALL 句 . . . 101
HACK 8.14 CASE 句 . . . 101
HACK 8.15 LIMIT 句 . . . 103

第 9 章 SQL to UNIX マイグレーション (3) – ソート・結合の句 105
HACK 9.1 ORDER BY 句 . . . 105
HACK 9.2 INNER JOIN 句 . . . 106
9.2.1 結合条件が複数列に及ぶ場合 . . . 109
9.2.2 行数が膨大であるなどの事情で、事前ソートが困難な場合 . . . 111
HACK 9.3 LEFT [OUTER] JOIN / RIGHT [OUTER] JOIN 句 . . . 114
9.3.1 結合条件が複数列に及ぶ場合 . . . 116
9.3.2 行数が膨大であるなどの事情で、事前ソートが困難な場合 . . . 116
HACK 9.4 FULL [OUTER] JOIN 句 . . . 117

HACK 9.5 CROSS OUTER JOIN 句 121

第 10 章 SQL to UNIX マイグレーション (4) – 関数 125
HACK 10.1 ABS 関数 125
HACK 10.2 ACOS 関数 125
HACK 10.3 ASIN 関数 126
HACK 10.4 ATAN / ATAN2 関数 127
HACK 10.5 AVG 関数 128
HACK 10.6 CAST / CONVERT 関数（データ型変換） 129
HACK 10.7 CEIL / CEILING 関数 130
HACK 10.8 COALESCE 関数 131
HACK 10.9 CONCAT 関数 132
HACK 10.10 CONVERT 関数（データ型変換） 133
HACK 10.11 CONVERT 関数（文字コード変換） 133
HACK 10.12 COS 関数 134
HACK 10.13 CURRENT_DATE / CURRENT_TIME / CURRENT_TIMESTAMP 関数 135
10.13.1 日時・時間の加減算はどうやるのか 136
HACK 10.14 DATEDIFF / DATADIFF_BIG 関数 138
HACK 10.15 DBMS_RANDOM パッケージ 138
HACK 10.16 EOMONTH 関数 138
HACK 10.17 EXP 関数 138
HACK 10.18 FLOOR 関数 138
HACK 10.19 IFNULL 関数 139
HACK 10.20 INITCAP 関数 139
10.20.1 全角アルファベットも先頭大文字化したい場合 141
HACK 10.21 ISNULL 関数 141
HACK 10.22 LAST_DAY / EOMONTH 関数 141
HACK 10.23 LCASE 関数 142
HACK 10.24 LEN / LENGTH / LENGTHB 関数 142
HACK 10.25 LEFT 関数 143
HACK 10.26 LN / LOG / LOG10 関数 143
HACK 10.27 LOWER 関数 144
10.27.1 全角アルファベットも小文字化したい場合 145
HACK 10.28 LTRIM 関数 146
HACK 10.29 MAX 関数 147

HACK 10.30 MID / MIDB 関数 148
HACK 10.31 MIN 関数 148
HACK 10.32 MOD 関数 149
HACK 10.33 NOW 関数 150
HACK 10.34 NVL 関数 150
HACK 10.35 PERIOD_DIFF 関数 150
HACK 10.36 RAND / RANDOM 関数 150
HACK 10.37 REGEXP_REPLACE 関数 151
HACK 10.38 REPLACE 関数 153
HACK 10.39 RIGHT 関数 153
HACK 10.40 ROUND 関数 153
HACK 10.41 RTRIM 関数 154
HACK 10.42 SIN 関数 154
HACK 10.43 SQR / SQRT 関数 154
HACK 10.44 SUBSTR / SUBSTRING / SUBSTRB 関数 155
HACK 10.45 SUM 関数 156
HACK 10.46 TAN 関数 158
HACK 10.47 TIMEDIFF / TIMESTAMPDIFF 関数 159
HACK 10.48 TRIM 関数 159
10.48.1 全角空白もあわせてトリミングしたい場合 160
HACK 10.49 TO_*関数（データ型変換） 160
HACK 10.50 UCASE 関数 160
HACK 10.51 UPPER 関数 161
10.51.1 全角アルファベットも大文字化したい場合 162

あとがき 163

第 1 章

File/Dir Hack 事始め

「インストール・設定が大変」、「バージョンアップが大変」、「データ移行・バックアップが大変」。そんな数々の苦労を強いられながら RDB 製品を使っているというなら、もうそんな苦労とはおさらば。さぁ、ファイルとディレクトリーと UNIX コマンドを使ったシンプルなデータ&タスク管理術「**File/Dir Hack**」を始めよう。

本章ではまず、何のためにこの File/Dir Hack を実践するのかを確認し、実践するためには不可欠な、切れ味よいコマンドセットを準備する。

HACK 1.1　まず、何を重要視するかを確認する

何をするにも目的意識が大事。目的なく実践しても得るものは無いし、目的を見失ってしまっては方針がブレるばかり。

ご存じのとおり、世の中には数々の RDB 製品がある。プロプライエタリーなものなら、Oracle、SQL Server、DB2……。オープンソースなものなら、MySQL、PostgreSQL、SQLite……。それらにはそれぞれ利点がある。その利点をよくわかっていて、あなたにとって必要不可欠であるのなら、それらを使うべきだ。しかし、さほど必要ではなかったり、あるいは欠点の方が上回っていて他の手段を検討したいというなら、これから記す話をよく聞いてもらいたい。

1.1.1　File/Dir Hack が目指すもの

答えを先に記すと、ずばり次の 3 つの性質である。

- 見やすさ・分かりやすさ
- 速さ
- 強靭さ

後続の HACK で解説していこう。

HACK 1.2 見やすさ・分かりやすさを追求する

読み書きに無駄なコストを掛けない。データはテキストで保存し、かつ、一目で分かる内容を心掛ける。

見やすさ・分かりやすさは、データにとって最も重要なものだと考えている。何かの暗号文があったとして、それはあえて分かりにくい姿にされているが、平文に比べて解読に時間が掛かる。通常のデータでこのように読み込みにコストを掛ける理由などないから、見やすく・分かりやすくあるべきだ。

1.2.1 データはテキストファイルに保管する

マイク・ガンカーズの唱える UNIX 哲学[*1]の定理 5 に

> Store data in flat text files.
> 単純なテキストファイルにデータを格納せよ。

と記されているように、見やすさ・分かりやすさを追求するのなら、第一に、データはバイナリーファイルではなくテキストファイルに格納すべきだ。

世の RDB 製品の多くは、データをバイナリーファイルとして格納しており、それを開くには、その RDB 製品専用のソフトウェアが必要であるが、専用のソフトウェアを動かすことも、使い方を覚える事も、コストに繋がる。避けられるならそれに越したことはない。一方、テキストファイルに格納しておけば、**どの環境にでもあり（= 追加インストールの必要もない）、かつ極めて汎用的な cat や less コマンド等で素早く開ける。**

1.2.2 一発で理解できる見た目を心掛ける

cat や less コマンドなどで素早く開けても、難解な姿をしていては意味がない。難解であるならバイナリーファイルを使う方がましなくらいだ。

それらの単純なコマンドで開いたら、誰でも即内容を理解できることが望ましい。コンピューターでデータを扱う場合には、人間のみならず、コンピューターにとっての扱い易さも重要であるが、まず人間にとっての見やすさが第一だと我々は考える。開発・保守を行う人間の作業効率が上がるし、データを処理するコンピューターのアルゴリズムも人間を基準に設計されているものが多く、結局は処理効率も人間の分かりやすさに影響されがちだからだ。

見やすく・分かりやすいテキストファイルとは?

ここで、次のテキストファイルを見てもらいたい。

■20181 学期中間テスト.txt

```
        :
17B2430511000200490004
```

[*1] オーム社のロングセラー書籍「UNIX という考え方」を購読されたい。

```
17B244064060020004048
17B245002016005008009
17C101070100091055033
17C102058003078098017
          :
```

ファイル名に“20181学期中間テスト.txt”と書いてあるが、これはある学校の全生徒（学籍番号 6 桁）の英語・数学・国語・理科・社会の 5 教科の成績データである。学籍番号は 6 桁で、テストは 0 点から 100 点までの各 3 桁ということでこういう仕様になっている。だが、そういう仕様だと説明されなければ分からないし、分かっても非常に見づらい。

テキストファイルなのだから、各列を固定長にせずに空白で区切れば見やすさが向上するはずだ。

■20181 学期中間テスト_1.txt

```
          :
17B243 51 100 20 49 4
17B244 64 60 20 4 48
17B245 2 16 5 8 9
17C101 70 100 91 55 33
17C102 58 3 78 98 17
          :
```

最初のものに比べるとだいぶ見やすくはなったが、得点の桁数によって各生徒の列がガタガタになっている。欲を言えば、各教科の列の位置は揃っていてもらいたい。

■20181 学期中間テスト_2.txt

```
          :
17B243  51 100  20  49   4
17B244  64  60  20   4  48
17B245   2  16   5   8   9
17C101  70 100  91  55  33
17C102  58   3  78  98  17
          :
```

見やすさの追求とは、こういった配慮である。

HACK 1.3　無理なく速さを追求する

どのようにデータを配置すればコンピューターの計算量は抑えられるのか、このことを常に意識する。人間にとって見づらくならない程度に。

二番目に重要なことは「速さ」。ここでいう速さとは、テキストデータ読み込みの速さや、データを加工する速さなど、コンピューターにとっての速さを指している。もちろん速いに越したことはないが、先ほど論じた見やすさと両立できない場合には、見やすさを優先すべきだと我々は考えている。ただし、**速さの追求は、システムの実用性を損なわない程度には必要**だし、コツをつかめば両立もさほど困難ではなくなる。

速さを引き出うえで本質的に必要なことは、取り扱うデータ量の削減と計算量・計算時間の削減であるが、それを実現するために具体的にできることは次のようなことである。

- データの持たせ方の工夫
  - 実名はマスターデータとして別ファイルにし、通常は文字数の少ないコードで扱う。
  - 1 つの行（レコード）にたくさんの列（カラムまたはフィールド）を持たせない。
  - 行が増えすぎてきたら、複数ファイル等での分割を考える。
  - 早引き表（インデックス）ファイルを作る。
- データ処理量・計算量の削減
  - 処理対象でないデータのフィルタリングをなるべく先に行う。
  - ソートの発生を抑える設計にする（ソートされた状態をなるべく維持する）。
- ハードウェアの更新
  - 速いコンピューターに替える（&そのために環境依存の少ない設計にしておく）。

ここで列挙した各々の配慮は、人間が手作業で行う場合にも言えるものが多い。つまり、コンピューターにとっての速さに繋がる工夫も、人間が直感的に想像できるものが多い。

HACK 1.4 強靭さ “robustness” を追求する

軽くない障害や事故、あるいは事件に巻き込まれても、復旧できる可能性が高く、しかも素早く復旧できるデータであることが重要。

我々が、他人の設計を見て「足りていない」としばしば思う要素、それは強靭さである。

高速・高性能・高機能を追求し、他との差別化を図るのは大いに結構だ。システムが正常に稼働している時はそれでうまくいく。しかし想定外の事態に見舞われた時、一体どうするつもりなのだろうかと甚だ疑問なのである。

例えばこのような事態が起こる可能性、あるいは起こったことは無いだろうか?

- ソフトウェア的な事故
  - 自ら制作したソフトや依存ソフトウェアのバグによるデータ破壊
  - 依存ソフトウェアの脆弱性がマルウェアや不正アクセスを許す
  - 運用者の操作ミスによるデータ喪失
- 物理的な事故
  - ディスクをはじめとしたハードウェアの故障
  - 施設の停電でデータが飛ぶ
- 時代の波
  - 依存ソフトウェアのサポートが終了した
  - 依存ソフトウェアの仕様が変わってしまった
  - 業務ルールが変わり、データを変換しなければ無駄になる
- 事件
  - 自然災害
  - 組織内に悪意を持った人物がいて、システムを壊しにくる

「ほとんど運用上の問題じゃないか! 開発者の考える事じゃない」とか「そんな事まで考えてたらキリないだろうが」などと嘲笑うかもしれない。しかし、運用者の立場に立って考えてもらいたい。このような想定外の事態に見舞われた時、それでもデータが無事だったら、いや、完全に無事とはいかなくても部分的にでも残っていてくれていたら……、涙が出るほど嬉しいだろう。**プログラムはまた作り直せるが、データは二度と復活できない**のである。しかもそういった事件・事故が発生した時は大抵切迫しており、一刻を争う

ことが多い。つまり、一秒のコストが物凄く高いのである。

ということで、想定外の事態に対して運用者が気を付けるのはもちろんのことだが、「想定外」とは想定できないからこそ想定外なのであり、すべて気を付けられるわけではない。そんな時に頼りになるソフトウェアの選択肢がないのはあまりにも酷いが、世の中を見回しても少なすぎるような気がする。ソフトウェアは、運用者に利益をもたらすために存在するのだから、ちゃんと異常事態のことも考えてあげなければ使命を果たせているとは言えない。

では、強靭さを得るために具体的にどんな対策をしておくべきか。データに関して言えばまず、**cat や less、vi コマンド等で内容を把握できるようにしておくこと**だと我々は考える。想定外の事態が発生すると、システムに思わぬ障害を受ける可能性が高い。そんな時、高度なビュアーなしでは中身がわからないようなファイルフォーマットだったら、動いてくれないかもしれない。例え動くとしても、動かすまでに多くの時間をとられるかもしれない。緊急時は一刻を争うことがしばしばであるため、手間取るということはそれだけでも致命傷になり得る。cat、less、vi などといったごく最低限のコマンドで開いて中身が把握できるようになっていれば、助かる可能性が高い。また、これはテキストデータに一般的に言えることだが、データの一部が壊れたり文字化けしても、他の無事なデータは救える可能性が高い。一部が壊れただけで全体が救済不可能になるようなデータフォーマットにだけはしてはならない。

現場で散々痛い目に遭ってきた者が言うのだから間違いない!

HACK 1.5　データやディスク容量をケチらない

ディスク増設費用と、それをケチったばかりに起こる効率悪化や損害で被るコストでは、たぶん前者の方が圧倒的に安い。

データはバイナリー形式ではなくテキスト形式で保管するという方針を示した時に、

> でもそれだとデータサイズが膨れてディスクが無駄に消費され、コストが嵩む。

と思いはしなかっただろうか。たぶんその考え方は古いか、あなたが組み込みシステムなどのリソースが極端に限られた環境の開発者でなければ当てはまらない。

例えば 1980 年代だと数十 MB の HDD が数十万円、つまり 1MB が 1 万円もする時代だった。それが 1990 年代後半になると 1GB で 1 万円になり、2010 年代には 1TB で 1 万円を切った。つまり、この間に同じ予算で 100 万倍の容量が手に入るようになった。では、あなたが作ろうとしているシステムのデータはこの勢いで肥大化しているのだろうか。「私は世界のビッグデータを扱っています」とか「私は大量の動画データを管理しなくてはなりません」という一部の例外を除き、大半の人はそうではないだろう。管理すべきテキストデータがたくさんあったとしても、並大抵の努力ではディスクを埋め尽くせるほど膨大にはならない。

こんな時代になった今、ディスク容量をケチるのはまったくのナンセンスだ。ディスク容量をケチってコストを抑えることを考えるくらいなら、そのデータが消えた時にどれだけ膨大なコストが発生するかを想像し、それよりもずっと安上がりなはずのディスクをもう一台買って、今すぐバックアップをとるべきだ。

悪いことは言わない。

> **システムを入れるディスクの容量は、今必要だと見積もられる量の 3 倍以上のものにしておくべきだ。**

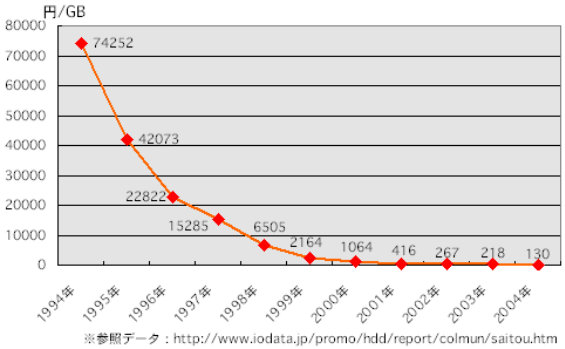


図 1.1 HDD 容量単価の推移 出典:http://pc.nf4hou.com/hardware/hdd_kakaku.html

当初の見積もり以上にデータが増えたり、システム改修でデータ項目が増えたりといったことが往々にして起こってきたし、そして、この先の HACK においても、効率化を目的としてあなたの想像以上にディスク領域を使うテクニックがいくつか出てくる。そいう時でも、経験的に 3 倍の余裕を見ておけば、差し迫った問題に遭遇することはまずなかったからだ。

1.5.1 ただし、目的無き消費は「無駄」である

もちろん、ディスク領域を無制限に消費することを推奨するわけではない。ディスクに対するコストはさほど気にしなくてよい時代になったとはいえ、闇雲にデータサイズを増やせば、HACK 1.1 で目指すとした性質のうち、特に

- 見やすさ・分かりやすさ
- 速さ

が失われてしまうだろう。無意味なデータが増えれば目的のデータがどこにあるか分かりにくくなるし、読み書きにも時間がかかるので効率が悪化する原因になってしまう。

データをバイナリー形式ではなくテキスト形式で持つとしたことにも、見やすさ・分かりやすさ、さらには基本的 UNIX コマンドでの扱いやすさといった目的があってのことだ。

何らかの目的無しにデータが増えていくのは「ケチらない」のではなく「無駄」なので注意すること。

HACK 1.6 システムを擬人化し、手作業時代の構造を写像する

「もしコンピューターが人だったらどうやるだろうか」と考える。

コンピューターはある意味凄い。人間には到底真似できない記憶力と計算力、そして持久力を持っている。しかし、それだけである。人間がやってきた作業を真似して、その作業を人間以上の効率でできるようになっているに過ぎない。これは今のところ AI であっても言える。もはや将棋は人間以上に強いだろうが、他のテーブルゲームを勝手に覚えて次々制覇していくことはできない。

つまり、コンピューターを難しく考えるなということだ。何か高度な機能を覚え、使って、魔法のように

仕事をやらせる相手ではない。コンピューターのことは、「確かに働き者だけど、言われたことしかできない新人アルバイト」くらいに考えた方がいい。

1.6.1　ショッピングカートプログラムが人だったら

どういうことか理解するために、ショッピングカートプログラムを例にして考えてみよう。

これはモノを売る「店」を自動化しているのだから、店にはどんな役の人がいるかを考えればいい。商品を陳列する人、売り場案内をする人、レジで会計をする人、裏で在庫管理をする人、などの役がある。さらには、商品を棚から取ってレジへ持っていく客も役と言える。

従って、これらの役単位でプログラムを作り、連携させていくことを考えるのである。それぞれの役が具体的にどんな作業をしているかを細かく洗い出していって、プログラムに起こしていけば完成である。こうして見れば、「プログラムに起こす」という作業は人がやっていた作業をプログラムに写像することである。

1.6.2　書類や収納棚のデータ構造をファイル・ディレクトリーに写像する

写像するべきは、人の作業だけではない。店には商品管理や注文管理、売上管理のための帳簿や伝票があるはずだ。人の作業を写像しようとしているのだから、人が扱っているこれらもデータもセットで写像しなければ意味がない。この時もやはり、「コンピューターにはデータ管理のための高度な概念があって……」などと難しく考えないことだ。

例えば、ある帳簿が 10 列で構成された表形式になっているのであれば 10 列の SSV 形式[*2]のファイルを作る。その帳簿が冊子になっているというならば、その冊子に相当するファイルを格納するディレクトリーを作る。さらにその冊子が、年度のラベルが貼られた引き出しにしまわれているならば、年度の名前を付けたディレクトリーにそのディレクトリーを入れる、……といった具合に。もちろん、コンピューターでは扱いづらい構造であると思える箇所がもしあれば、適宜アレンジすればいい。

図 1.2 手作業時代の構造とは、帳簿のレイアウトや書類棚の構造そのもの

[*2] カンマ区切りの CSV ならぬ、スペース記号区切りのデータに対する本書での呼称であり、“`/etc/fstab`” や “`/etc/crontab`” などが典型例。詳細は HACK 2.6 にて。

このようにして、既存の作業やデータ構造をほぼそのまま写像すれば、当初の目的であった「見やすさ・分かりやすさ」にも大いに貢献する。

HACK 1.7 File/Dir Hack 用コマンドセット “ShellShoccar” のインストール

どの UNIX ホストにもある基本的 UNIX コマンドの機能面での貧弱さを埋めるパワフルなコマンドセット。File/Dir Hack の実践でも威力を発揮する。

さてこの後、File/Dir Hack の実践に入る。

1.7.1 コマンドセット “ShellShoccar” とは何か

コマンドセット “ShellShoccar” は、我々「秘密結社シェルショッカー」が開発したりまたは移植したコマンドのコレクションである。その目的は、シェルスクリプトによるシステム開発を実用的な選択肢にすることだ。

開発言語としてシェルスクリプトを選択することで獲得できる他に類を見ない特長は、

- 今や Windows、Mac、UNIX と、一度書いたらどこでも動かせる高い互換性
- 同じく、一度書いたら 10 年、20 年もの長きに渡り、メンテナンスフリーで使い続けられる持続性

である。

移植元のコマンドとしては、POSIX 標準ではないが多くの OS にある便利なコマンド（seq など）や、Open usp Tukubai[*3] と呼ばれるシェルスクリプト開発向けのパワフルなコマンド群の一部が含まれている。

コラム

POSIX 原理主義

実際には、単にシェルスクリプトを使うのみでは、高い互換性や持続性は得られない。それらの性質を獲得するには、開発言語としてシェルスクリプトを選択したうえで、**POSIX 原理主義**と呼ばれるプログラミング作法を適用しなければならない。

POSIX 原理主義とは、RAID や電源二重化のように、プログラムコードに冗長性を持たせることで、互換性・持続性を高める指針である。

詳しくは、C&R 研究所から発行されている書籍「Windows/Mac/UNIX すべてで 20 年動くプログラムはどう書くべきか」を参照してもらいたい。

しかし、基本的 UNIX コマンド（どの UNIX 系環境にも初めから入っているコマンド）だけでは、いまいち機能が貧弱だと感じざるを得ない場面が多々ある。システム開発で頻繁に登場する「典型的な処理」が簡単には書けないのである。そこで、そういった処理（機能）をコマンド一発で実現できるようにしたものがこのコマンドセットだ。

本書で解説する File/Dir Hack にも、それらコマンドでカバーされている「典型的な処理」がやはり頻

[*3] `https://uec.usp-lab.com/tukubai`

繁に登場するため、ほぼ必須のコマンドセットといえる。とはいっても各コマンドの実体は、内部で基本的 UNIX コマンドを適宜呼び出しているシェルスクリプトに過ぎず、いわばラッパーなので、互換性や持続性という当初の特長を失わせるものではない。

また、ライセンスもパブリックドメイン（CC0）とし、互換性・持続性が法的にも阻害されないように配慮している。

1.7.2　コマンドセット “ShellShoccar” をインストール

インストール作業は簡単だ。基本的には我々が次のサイト

```
https://github.com/ShellShoccar-jpn/
```

で公開しているコマンド群を、1) ダウンロード、2) 実行ビット立て、3) パスを通す、をすればよいのだが、これらを自動化するためのシェルスクリプトを用意してあるのでそれを実行するのが手っ取り早い。ただし、(a)git コマンド、(b)curl と unzip コマンド、(c)wget と unzip コマンドの、(a)(b)(c) いずれかの組み合わでコマンドを用意する必要があるので、もし無ければ揃えておいてもらいたい。

準備ができたら、まず次のようにしてインストーラーをダウンロードする。

```
（wget コマンドがある場合）
$ wget https://raw.githubusercontent.com/ShellShoccar-jpn/installer/master/shell
shoccar.sh↵
$

（curl コマンドがある場合）
$ curl -o https://raw.githubusercontent.com/ShellShoccar-jpn/installer/master/sh
ellshoccar.sh↵
$
```

そして、インストーラーを実行する。管理者としてインストールする権限がない場合には、後者のようにして自分のホームディレクトリーにインストールできる。

```
（管理者としてインストールする場合→ “/usr/local/shellshoccar” に入る）
$ sh shellshoccar.sh install↵
$

（一般ユーザーとして自分のディレクトリー内にインストールする場合）
$ sh shellshoccar.sh --prefix=~/shellshoccar install↵
$
```

インストールが成功すれば、最後のメッセージの中で、環境変数 PATH に追記すべきディレクトリーが表示される。普段使っているシェルの初期設定ファイル

- 管理者としてインストールしたなら、恐らく次のファイルのうち、PATH の定義が記載されているもの

  - “/etc/profile”
  - “/etc/*shrc”
- 一般ユーザーとしてインストールしたなら、恐らく次のファイルのうち、PATH の定義が記載されているもの
  - “~/.profile”
  - “~/.*shrc”

をテキストエディターで開いて、その中で記述されている PATH の定義箇所に、今指示されたディレクトリーを追加する。また、その変更が反映されるように、（source コマンドなどで）今編集したファイルの読み込みもしておく。

最後に、ちゃんとインストールがされたかを確認するために、収録されているコマンドの一つである “urlencode” を次のように実行してみてもらいたい。

```
$ urlencode --help↵
Usage   : urlencode [-r|--raw] <file> ...
Args    : <file> ..... Text file for URL encoding
Options : -r, --raw ... RAW MODE :
                        " " will not be converted into "+" but "%20"
Version : 2017-07-18 02:39:39 JST
          (POSIX Bourne Shell/POSIX commands)
$
```

Usage の最後に “(POSIX Bourne Shell/POSIX commands)” と表示されていれば成功だ。ちなみに、このコマンドはその名前のとおり、到来した文字列を URL エンコードするためのコマンドである。

第 2 章

データ管理のためのファイルの使い方

「データはテキストファイルに格納する」という方針を定めたものの、実際に手を動かしてみると、これまで RDB 製品に慣れ親しんでいた者にとっては、「ええと、こういう場合どうすればいいの?」といろいろな手順で疑問が湧くだろう。

同じ道を辿ってきた我々が、数々の経験に基づき見つけてきたベストな答えを、ここにまとめる。

最初に、テキストファイルに書き入れる文字列に関するルールを解説し、次に、そのルールに基づいて決めたファイルフォーマットについて紹介する。

HACK 2.1　「列区切りは、半角空白 1 個以上」と決める

世の中には CSV、タブ区切り、コロン区切りなど、様々あるが、任意の長さの空白で区切るのが一番見やすい。UNIX のコマンドも設定ファイルも、そう作られている。

紙の書類でもコンピューターでも、一般的に管理がなされているデータは、「見出し」と、それに紐づけられた「データ本体（1 つとは限らない）」という構造をとっている。そして、その構造を表現するために最も頻繁に用いられている形態は、恐らく行と列から構成された二次元の「表（テーブル）」である。

コンピューターのテキストファイルで表を表現するにあたって、最初に決めなければならない事項の一つは、「列区切り」である。

もちろん行の区切りも決めなければならないが、これは UNIX を使う場合にはほぼ LF（0x0D）と決められるのでさほど悩むことはない。一方で列区切りは、例えば CSV ファイルではカンマ記号（“,”）が使われ、“`/etc/passwd`” ファイル等ではコロン（“:”）が使われ、その他多くの UNIX 設定ファイルでは半角空白記号（の 1 文字以上の連続）が使われている。さて、どの方式を採用すべきか。

しかし、この HACK の題名にも記したように、（File/Dir Hack のためには）「列区切りは、半角空白 1 個以上」にすべきであると、我々は結論づけた。

2.1.1　CSV じゃダメなのか?

現在、テキストファイルへのデータ格納形式としては最も普及しているのは、恐らく CSV ファイルだろうと思う。CSV 形式は歴史も長く、それでいて RFC 4180 という規格により細部まで仕様が統一されており、互換性の面でも優位な立場にある。また RFC 4180 仕様では、通常あり得るすべてテキスト文書を矛

盾なく格納できる[*1]。

それなら CSV フォーマットを採用すればよいように思う。しかし、そう簡単にはいかない問題があった。sed、AWK をはじめとする基本的 UNIX コマンドでは簡単に扱えないのだ。

例えば次の CSV ファイルがある。1 列目から順に、番号、氏名、自己紹介文という 3 列で構成されている。

■自己紹介.csv

```
1,井川 さくら,"1995年生まれの秋田出身です。
新参者ですがよろしくお願いします。"
2,千葉 みなと,"1986年生まれ。出身は苗字のとおりです。
去年はだいたい一日16,733人くらいの人と会いました。"
3,武 豊,"明治19年生まれ。出身は愛知県です。
""たけゆたか""ではなくて""たけとよ""です。"
```

この中から 2 行目にある 3 列目の文字列が欲しいと言われたら、それら基本的 UNIX コマンドでは簡単には取り出せない。行と列の区切り文字であるはずの改行とカンマが文字列に含まれており、区別が難しいからだ。

また、漠然とではあるが、見づらい。

これが CSV を採用しなかった理由だ。

2.1.2 「半角空白 1 個以上区切り」で、見づらさを解消する

そこで、半角空白 1 個以上区切りを採用していくことになるのだが、これにより、まず CSV にあった隣り合う列の見づらさが解消できる。

先程の、自己紹介 CSV の例はひとまず置いておいて、メールアドレスを横に書き並べた次の (1)〜(3) の例を見比べてもらいたい。

(1) カンマ区切りの場合

```
s.igawa@example.com,m.chiba@example.or.jp,m.iwaki@example.ac.jp
m.omi@example.jp,t.take@example.jp,m.urawa@example.jp
```

(2) 半角空白 1 文字区切りの場合

```
s.igawa@example.com m.chiba@example.or.jp m.iwaki@example.ac.jp
m.omi@example.jp t.take@example.jp m.urawa@example.jp
```

(3) 半角空白 1 文字以上区切りの場合

```
s.igawa@example.com m.chiba@example.or.jp m.iwaki@example.ac.jp
m.omi@example.jp    t.take@example.jp     m.urawa@example.jp
```

(1) のようにカンマ区切りで並べたものは、カンマがメールアドレス文字列中のドット記号と似ていて見づらい。では最も見やすい代替文字は何かと考えれば、(2) のように空白（半角空白）が適していることが

[*1] 行を区切る改行文字や列を区切るカンマが文字列データとして含まれている場合も、しっかり区別できる仕様であるという意味。

わかる。さらに、(3) のようにして桁揃えをすれば一層すっきりする。

ただし、File/Dir Hack では、**(2) のようにするか (3) のようにするかは任意**とする。vi エディター等を使って手で編集するなら簡単に (3) のようにできるので、そうやって可読性を上げればいい。しかしながら、AWK コマンドなどのプログラムから書き出す場合には桁揃えが面倒なので、(2) のようにしてもよいものとする。コマンドに読み込ませるぶんにはどちらでも問題ないし、前章の HACK 1.7 でインストールしたコマンドセットには、自動的に桁揃えしてくれる便利なコマンド “keta” が用意されているからだ。

keta コマンドで桁揃え

列の区切り方は (2) でもいいと決めたものの、そのままでは読みづらくて気になる場合もある。そんな時は、(2) のように桁の揃っていないテキストデータを、標準出力経由で “keta” というコマンド[*2]に流し込めば自動的に桁揃えをしてくれる。

■keta コマンドの使用例

```
（元のテキストデータは列が揃っていない）
$ cat mailaddr.txt↵
s.igawa@example.com m.chiba@example.or.jp m.iwaki@example.ac.jp
m.omi@example.jp t.take@example.jp m.urawa@example.jp
$
（keta コマンドで開くとデフォルトで右揃え (数字用) になる）
$ cat mailaddr.txt | keta↵
s.igawa@example.com m.chiba@example.or.jp m.iwaki@example.ac.jp
   m.omi@example.jp     t.take@example.jp    m.urawa@example.jp
$
（--オプションを付けると左揃えになる）
$ cat mailaddr.txt | keta --↵
s.igawa@example.com m.chiba@example.or.jp m.iwaki@example.ac.jp
m.omi@example.jp    t.take@example.jp     m.urawa@example.jp
$
```

これで、(2) の生成しやすさと (3) の見やすさを両立できる。簡単に (3) のように揃えられるので、プログラム中で keta コマンドを使って常に桁の揃ったファイルを出力してもいいが、すぐ揃えられるのだから (2) のままでいい。コンピューターにとっては (2) の状態の方が若干、処理効率もいい。

なお、データは標準出力から与えずとも、ファイル名として引数から与えてもよい。

[*2] Open usp Tukubai からの移植コマンド。詳細は次の URL を参照。
https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_keta

2.1.3 「半角空白 1 個以上区切り」は、UNIX コマンドにとっても都合がいい

列区切りとして半角空白 1 個以上というルールを採用した最大の理由は、基本的 UNIX コマンドがどれもデフォルトでそうなっているからである[*3]。デフォルトに従っておく方がそれら基本的 UNIX コマンドを利用するうえでの記述もシンプルで済み、生産性や可読性が上がる。「郷に入っては郷に従え」ということだ。

そもそも、なぜ基本的 UNIX コマンドのデフォルトがそうなっているのかと想像を巡らせれば、きっと先程のメールアドレス横並びの例 (3) が、UNIX 発明者たちにとって最も見やすかったからだと思う。実際、/etc ディレクトリーの中の設定ファイルも、タブ文字（0x09）も交ぜながら、列揃えされているものが多い[*4]。

話の流れから、ここで大事なことをもう一つ述べるが、**半角空白文字（0x20）とタブ文字（0x09）はどちらも好きに使ってよい。**基本的 UNIX コマンドのデフォルトがそれを許容しており、実在の設定ファイルもそうなっているからだ。

というわけで、File/Dir Hack における列区切りについてまとめる。

【決め事 1】列の区切り方

列は、半角空白（0x20）1 個以上の連続で区切る。
水平タブ（0x09）1 個以上の連続の使用も認める。

半角空白は、1 個でも、あるいは 2 個以上連続していても、隣り合う 2 つの列の区切りとして等価（CSV の場合のカンマ 1 個に相当）であるという意味だ。また、その次の文章は、半角空白文字と水平タブも全く等価に扱われるという意味だ。

2.1.4 値として半角空白・タブを含ませたい場合はどうするの？

CSV ファイルの問題点を指摘した際に、列区切りとしてのカンマと、値（文字列）としてのカンマを区別するためのルールが複雑で、sed や AWK で簡単に書けないということを言った。

この問題は、半角空白区切りにしたところで解決できないし、どんな記号を区切り文字に採用しても、その記号が文字列中に含まれれば同じ問題が生じてしまう。

これについては次の HACK で解決する。

HACK 2.2 文字列中の半角空白は、“_”（アンダースコア）で表す

半角空白のように見た目を邪魔せず、それでいて半角空白文字と見た目の区別がつくアンダースコア記号“_”が、経験的に最も適した代替記号だった。

【決め事 1】では列区切に半角空白文字を使うこととしたため、文字列中にもともとあった半角空白文字をどうにかしなければならない。そこで新たな決め事を設ける。

[*3] じつは sort コマンドだけはデフォルトが違うのだが、“-b” オプションで同じにできる。

[*4] 例えば、fstab や crontab、resolv.conf あたりである。

【決め事 2】文字列中の半角空白の表現

文字列中の半角空白は、“_”（アンダースコア）で表現する。
同様に、アンダースコアは、手前にバックスラッシュを付けて（“_”）エスケープする。

※ “\” 自身のエスケープに関する決め事は【決め事 3】で。

この決め事には否定的な意見もよく出される。「なぜわざわざ、素のアンダースコア記号に特殊な意味を持たせるのか? エスケープ記号として一般的なバックスラッシュを使い、半角空白の方を “_” とすればいいのに」と。

その理由は、これまでの現場経験から、

- 見た目を極力邪魔しないでほしい。（バックスラッシュ記号が入ると見た目が邪魔で、空白らしくなくなる）
- 「バックスラッシュ記号 + 代替文字」として 2 文字にすると、横幅が変わってしまう。
- 必ず何かの文字をエスケープしなければならず、横幅を完全に維持はできないが、それなら、より出現頻度の少ないアンダースコアの方をエスケープすべき。

という考えに行きついたからだった。

具体的どういう見た目になるかは、次の HACK の後でまとめて例示する。

HACK 2.3　改行・タブ等のその他特殊文字はバックスラッシュ記号でエスケープする

printf のフォーマット記述と同じルールにしてしまえば、printf で簡単に元に戻せる。

「半角空白以外にも区切り文字としての仕様を認めているタブ（0x09）を文字列に含めたい場合はどうするのか」、「改行記号（0x0D や 0x0A）、さらにはエスケープ用のバックスラッシュ自身を文字列に含めたい場合はどうするのか」などの問題が未解決であるため、さらに決め事を設ける。

【決め事 3】その他特殊文字の表現

タブ、改行、バックスラッシュ自身等の特殊な文字は、printf と同様のルールで表現する。すなわちそれぞれ、“\t”、“\n”、“\\” である。

ここでは明記はしなかったが、printf と同様のルールなので、エスケープしうる文字は次の表のとおりである。

対象文字	ASCII コード	表現方法	補足
\	0x5C	\\	ほぼ対応必須
HT（タブ）	0x09	\t	ほぼ対応必須
LF（改行）	0x0A	\n	ほぼ対応必須
CR（行頭復帰）	0x0D	\r	稀に対応必要
BEL（ビープ音）	0x07	\a	殆ど対応不要
BS（一字戻る）	0x08	\b	殆ど対応不要
FF（改ページ）	0x0C	\f	殆ど対応不要
VT（垂直タブ）	0x0B	\v	殆ど対応不要

ただし、後半の方はほとんど使わない文字だろうから、エスケープ・アンエスケープ処理を常に真面目に実装しなければならないというわけではない。生産効率や処理能力が問題になるようならむしろ省略すべきだろう。

2.3.1 エスケープ・アンエスケープのコード例

今後、ここで取り決めたルールに従ってエスケープ処理をする場合、反対にそのようにして格納されているデータファイルから文字列を取り出して元に戻す場合、それぞれの場合に書くことになるプログラムコードを例示する。

■エスケープ処理

```
cat TARGET_DATA | # ←エスケープした文字列データを標準入力に流す
awk '
  BEGIN{
    s="";
    while(getline l){s=s l "\021";} # 1)各行をダミー文字を挟んで結合
    s=substr(s,1,length(s)-1);      #    (末端のダミー文字は不要)
    gsub(/\\/  ,"\022",s);          # 2)"\"を一旦ダミー文字2に退避
    gsub(/_/   ,"\\_" ,s);          # 3)"_" → "\_"
    gsub(/ /   ,"_"   ,s);          # 4)" " → "_"
    gsub(/\t/  ,"\\t" ,s);          # 5)タブ→ "\t"
    #
    gsub(/\r/  ,"\\r" ,s);          # 6)この部分は
    gsub(/\a/  ,"\\a" ,s);          # 7)通常は
    gsub(/\b/  ,"\\b" ,s);          # 8)省略しても
    gsub(/\f/  ,"\\f" ,s);          # 9)影響は
    gsub(/\v/  ,"\\v" ,s);          #10)ない
    #
    gsub(/\022/,"\\\\",s);          #11)ダミー文字2→"\\"
    gsub(/\021/,"\\n" ,s);          #12)ダミー文字 →"\n"

    # この時点で、変数sにエスケープ済の文字列が
    # 入っているので、各種処理に使うなり、
    # そのままprintするなりすればよい。
  }'
```

■アンエスケープ処理

```
cat ESCAPED_FIELD_DATA | # ←アンエスケープした文字列データを標準入力に流す
awk '
  {
    s=$3; # 例えば3列目にエスケープ文字列が格納されているとする
```

```
    gsub(/\\\\/,"\021",s); # 1)"\\"を一旦ダミー文字に退避
    gsub(/_/   ," "   ,s); # 2)"_" →" "
    gsub(/\\_/ ,"_"   ,s); # 3)"\_"→"_"
    gsub(/\\t/ ,"\t"  ,s); # 4)"\t"→タブ
    #
    gsub(/\\r/ ,"\r"  ,s); # 5)この部分は
    gsub(/\\a/ ,"\a"  ,s); # 6)通常は
    gsub(/\\b/ ,"\b"  ,s); # 7)省略しても
    gsub(/\\f/ ,"\f"  ,s); # 8)影響は
    gsub(/\\v/ ,"\v"  ,s); # 9)ない
    #
    gsub(/\\n/ ,"\n"  ,s); #10)"\n"→改行
    gsub(/\021/,"\\"  ,s); #11)退避していたダミー文字を"\"に置換

    # この時点で、変数sにアンエスケープ済の文字列が
    # 入っているので、各種処理に使うなり、
    # そのままprintするなりすればよい。
  }'
```

エスケープ・アンエスケープ処理をしたい箇所で、これらのコードをスニペットとして利用すればよい。

また、普段はめったに取り扱わない “\a”、“\b”、……等も記述してあるが、これから取り扱おうとしているデータには含まれないとわかっているなら不要である。もちろん、半角空白（“ ”）やアンダースコア（“_”）なども含まれていないのであれば省略して構わない。

HACK 2.4　NULL 値は、“-” や “*” などの記号 1 文字で表す

この決め事に満足できないというなら、あなたのデータ設計はたぶん適切ではない。

【決め事 1】で、列区切りには半角空白 1 個「以上」を用いることとした。これは、基本的 UNIX コマンドのほとんどがそれを前提としたフォーマットのテキストデータを受け入れるようにしたので、それに合わせたというのが大きな理由であった。しかし、一つ大きな問題が発生する。NULL 値（空文字、0 バイト文字列のこと）が表せないということである。

もし、列区切りが半角空白 1 文字のみと取り決められているなら、半角空白が 2 文字連続している箇所にはそこに NULL 値があると断定できる。だが、1 文字でも 2 文字でも 1 つの列区切りの役目を果たせるルールにしたことにより、「(列区切り)+NULL 値 +(列区切り)」なのか「(列区切り)」なのか区別できないのである。

そこで、さらに新たな決め事を設ける。

【決め事 4】NULL 値の表現

NULL 値は、“-”（ハイフン）または “*”（アスタリスク）の一文字で表現する。
どちらも都合が悪い場合には、制御文字の US（0x1F）一文字で表現する。

これはつまり、どこかの列を読み込んでみて、それが “-”1 文字であったら NULL 値と見なせという意

味であると同時に、NULL 値を表現したい場合には、その列に “-” という 1 文字を格納せよという意味である。もし “-”1 文字だと都合が悪いデータの場合には、代わりに “*” を使う。この取り決めは、UNIX の伝統的な設定ファイル（/etc/fstab や/etc/crontab）にならったものである。

しかしながら、“-” だろうが “*” だろうが、あるいは他の文字を使うにしても都合が悪いという場合もあるだろう。例えば、ユーザー自由記述欄の文字列を格納したい場合など、どんな文字列が到来するかまったく予想がつかないような場合だ。そんな時は、通常のテキストデータとしては用いられることのない制御文字のうち、無難なもの[*5]を使えばよい。そのうちの一つが US（0x1F）というわけだ。

どうやって使うかというと、次のように printf コマンドでシェル変数に予め代入しておいて使う。もし AWK の中で使いたいなら “\037” という即値で指定できる。

■US（0x1F）の使用例

```
■シェルスクリプトの場合

US=$(printf '\037') # 予めUS文字を変数に入れておく

# A.2列目がNULL値であることを示したデータを書き出す例
echo "hoge $US piyo" >> hoge.txt

# B.変数recordの文字列の2列目がNULL値かどうかを判定する例
set -- $record      # $1,$2,$3,… に各列の文字列を格納する
if [ $2 = "$US" ]; then
  echo '2列目はNULLです。'
fi


■AWKの場合

# A.2列目がNULL値であることを示したデータを書き出す例
awk 'BEGIN {print "hoge", \037, "piyo";}'

# B.各列の2列目がNULL値である行を表示
awk '$2=="^037" {print;}'
```

ちなみに、「NULL 値を表したいなら NUL(0x00) が最も相応しいのでは？」と思うかもしれないが、**NUL(0x00) コードはテキストデータ処理には絶対に使ってはならない。**NUL コードは、C 言語で文字列終端を表すのに用いられてきたという歴史的経緯により、シェル変数に代入したり AWK で処理させようとすると誤動作してしまう[*6]。残念ながら、シェルスクリプトや AWK では NUL コードは扱えないのだ。

[*5] 次に記す、難ありなものを避ければよい。NUL(0x00), BEL(0x07), 0x08(BS), HT(0x09), LF(0x0A), VT(0x0B), FF(0x0C), CR(0x0D), EOF(0x1A), ESC(0x1B), DEL(0x7F)

[*6] 誤動作しない実装もあるが、誤動作するものの方が一般的なので、たとえ誤動作しなくても利用すべきではない。

2.4.1　NULL 値に頼るのは「設計の敗北」である

ここまで言っておいてなんだが、一般的に、データベースの設計書に NULL 値が登場するようなものは、ダメな設計である[*7]。RDB 製品で、テーブル上のカラムに NULL 値を許容するものを含めたり、そして実際に NULL 値を含むレコードが大量に存在すると、処理速度が劇的に低下するものが多いからである。

だから大抵は、NULL 値を認めないようにテーブル設定をし、レコード新規作成時に何らかのデフォルト文字列を設定しておく。シェルスクリプトでのテキストデータは、NULL 値を表現するために “-” 等の何らかの文字を使わねばならないが、これはいわば、RDB 製品でいうところの NULL 値禁止設定を強制しているようなものと言えるだろう。

HACK 2.5　ファイルフォーマットの種類を把握する

覚えるべきは、ネイティブテキストデータファイルの 2 種類と、その派生物 3 種だけ。

文字列に関する決め事の決着が付けば、今度はそれを格納する箱であるファイルのフォーマットについての決め事ができるようになる。

実際に、普段我々がどのようなフォーマットのファイルを利用しているかを最初にまとめておくと、次のとおりである。

```
[ファイル]
|
+-- [ネイティブテキストデータファイル]
|     |
|     +-- ●SSV形式
|     +-- ●key-value形式
|          |
|          +-- ○JSONPath-value形式
|          +-- ○XPath-value形式
|          +-- ○CSVindex-value形式
|
+-- [その他テキストファイル]
|     |
|     +-- JSON, XML, CSV等
|     +-- HTML, CSS, JavaScript等
|     +-- (その他)
|
+-- [バイナリーファイル]
      |
      +-- 画像ファイル
      +-- PDFファイル
```

[*7] ただし RDB 製品上では、NULL 値と空文字は区別され、後者は問題ない。

+--（その他）

本書はシェルスクリプトによるシステム開発のためのデータファイルの在り方について説明する本であるので、この図のうち、●や○印を付けた**ネイティブテキストデータファイル**（UNIX やシェルスクリプトにとってネイティブなデータファイル）についての解説をしていく。

ただし、画像や PDF ファイル等そのままのフォーマットで管理せざるを得ないものや、HTML・JSON・CSV 等、外部システムとの入出力を行う上で、その形式での読み書きを求められる場面もあるため、全く取り扱わないわけではないという意味で、上図にはそれらも含めている。

HACK 2.6 SSV 形式（広義の field 形式）

カンマ区切りファイルが CSV なので、その命名規則に基づけば、半角空白 (space) 区切りの形式は SSV だろう。

我々が取り扱うテキストデータファイルの中で最も基本的なものはこの、SSV 形式と称するものである。【決め事 1】で列区切りには半角空白 1 個以上を用いると決めたが、それに従って文字列を行・列に並べたファイルである。カンマ区切りデータファイルを「CSV ファイル」（Comma Separated Values）、タブ区切りデータファイルを「TSV ファイル」（Tab Separated Values）と呼ぶというルールにならい、「SSV ファイル」（Space Separated Values）とした。

【決め事 5】SSV ファイル

カンマ区切りの CSV、タブ区切りの TSV の類型で、半角空白 1 個以上の連続で列を区切ったファイルを **SSV ファイル**と名づける。

なお、改行コードは基本的に LF のみとする。基本的 UNIX コマンドのほとんどがそうだからである。よって、行末に CR が付いていたら最終列の文字列の一部と見なされるので注意すること。

さて、SSV の具体例としては HACK 2.1 の (2),(3) がちょうどよいので、再掲する。

■SSV 形式の例 (1)(=field 形式)

```
s.igawa@example.com m.chiba@example.or.jp m.iwaki@example.ac.jp
m.omi@example.jp t.take@example.jp m.urawa@example.jp
```

■SSV 形式の例 (2)

```
s.igawa@example.com m.chiba@example.or.jp m.iwaki@example.ac.jp
m.omi@example.jp    t.take@example.jp     m.urawa@example.jp
```

(1) は半角空白 1 文字で区切ってあるもの、(2) は読み易さのため半角空白を複数個連続させて各列の開始位置（横）を揃えているものである。

(1) に「field 形式」という名称を付記したが、ここで field 形式と呼ばれる SSV の類似ファイルフォーマットを頭に留めてもらいたい。これは、**半角空白 1 文字のみで区切られた**テキストデータファイルを指す。つまり SSV 形式のサブセットにあたる。field 形式は、頻繁にお世話になる Open usp Tukubai コマンド群が対象としている形式で、それらのマニュアルにも用語として頻繁に登場する。Tukubai コマンドでは公式には field 形式のみ対応を謳っているものの、実際には多くのコマンドが SSV 形式に対応している。

HACK 2.7　key-value 形式（name 形式）

1 つのキーに、1 つのバリュー（値）を並べただけのシンプルな構造。しかし、変数や配列・連想配列の代わりとして大いに使い道がある。

SSV 形式の派生フォーマットで、さらに「key-value 形式」というものを定義する。詳細は次のとおりだ。

【決め事 6】key-value ファイルの定義

SSV ファイルの派生で、次の仕様に従わせたものを **key-value ファイル**と定義する。

- 各行 2 列固定
- 列は半角空白 **1 文字**区切り
- 第 2 列は半角空白と NULL 値をそのまま表現（“_” でエスケープせず、“–” や “*” も用いない）

具体例として、都道府県コードをこの形式で表現すれば、次のようになる。

■key-value 形式による都道府県コード（一部省略）

```
1 北海道
2 青森県
3 岩手県
4 宮城県
5 秋田県
6 山形県
7 福島県
8 茨城県
9 栃木県
10 群馬県
11 埼玉県
    :
    :
47 沖縄県
```

key-value 形式の主な用途は、このように ID と名称が対になったマスターデータである。各行が 2 列に固定されていて、かつ、列区切りは半角空白 1 個と決まっているために、第 2 列には半角空白が含められるという柔軟性がある。また、NULL 値の表現にも代替文字を使う必要がない。

文字列に空白文字を含めてもよいという柔軟性と引き換えに、AWK では “$2” を指定することによる第 2 列の抽出ができなくなっている。しかし、列区切りが 1 文字固定という性質があるおかげで

第 2 列の先頭の文字位置 = 第 1 列の文字列長 +2

という性質があるため、第 2 列を取り出すことは難しくない。

■key-value 形式における第 2 列文字列の抽出例 (1)

```
cat <<-ADDRESS |
        machida.tokyo 東京都 町田市
```

```
        sumida.tokyo 東京都 墨田区
        ADDRESS
awk '# 第1列keyが"machida.tokyo"である行の第2列valueを取得する
     $1=="machida.tokyo" {
       f2=substr($0,length($1)+2);
       print f2;
     }'
```

とはいうものの、いちいちこのような AWK のスクリプトを書くのは面倒臭いし可読性も落ちるということで、Tukubai コマンドには上記の抽出を行ってくれる “nameread” コマンドというものがあって便利だ。

■key-value 形式における第 2 列文字列の抽出例 (2)

```
cat <<-ADDRESS |
        machida.tokyo 東京都 町田市
        sumida.tokyo 東京都 墨田区
        ADDRESS
# 第1列keyが"machida.tokyo"である行の第2列valueを取得する
nameread 'machida.tokyo'
```

なお、その Tukubai コマンドでは、key-value 形式のことを「name 形式」と呼んでいるので、記憶の片隅に置いておいてもらいたい。

2.7.1 join したくば 2 列固定の SSV 形式

key-value 形式は AWK では “$2” のようにして単純に第 2 列を抽出できないと書いたが、似たような理由により、key-value 形式は単純に join する（二つの表を内部結合・外部結合する）ことができない。“join” を始めとした join 系のコマンドは、列区切り（のデフォルト）が半角空白 1 文字以上になっており、空白を含む文字列や NULL 値を認識できないためである。

もし join したい場合には、予め（2 列固定の）SSV 形式、つまり半角空白や NULL 値を【決め事 2,4】に従うように変換しておく必要がある。

■key-value 形式の SSV 形式への変換例

```
cat <<-ADDRESS |
        machida.tokyo 東京都 町田市
        sumida.tokyo 東京都 墨田区
        ADDRESS
# ↓ここからSSV形式への変換処理
sed 's/_/\\_/g'       | # ←1)【決め事2】への対応
tr ' ' '_'            | #       :
sed 's/_/ /'          | #       :
sed 's/ $/ -/'          # ←2)【決め事4】への対応
cat > /tmp/cities.txt   # ←3)変換したものを保存
```

ただし、第 2 列には半角空白も NULL 値も存在しないことが明らかなもの（例えば先程の都道府県コー

ド）については当然ながらそのまま使える。また、join するという目的においては（可読性のために）列区切りの半角空白が 2 個以上連続していてもよい。

HACK 2.8　JSONPath-value 形式・XPath-value 形式

一見複雑に階層化された JSON や XML も、どの場所に、どんな値が入っているかを列挙していけば key-value 形式に変換できてしまう。

key-value 形式というものが定義できたことで、UNIX・シェルスクリプトの世界に一つ画期的な恩恵がもたらされる。それは、Web 業界のデータ交換に多用されている JSON や XML を key-value 形式で容易に表現可能になるということだ。

2.8.1　基本アイデア

UNIX で取り扱うデータは基本的に行と列から構成される表形式（二次元）であるのに対し、JSON や XML は階層構造をとっていて多次元である。しかしながら、格納されている各々の値には一意な場所が与えられている。ということは、場所と値を key と value に当てはめれば key-value 形式で表現できるはず、というのが基本アイデアである。

JSON も XML もテキストデータなのだから、key 部も value 部ももともと文字列であって容易に表現できる[*8]。そして都合がいいことに、JSON や XML における key 部、つまり場所を表現するにあたっては、JSONPath[*9]や XPath[*10]という標準ルールが存在するため、それに従えばよい。

ということで、以下の 2 つのファイルフォーマットを新たに定義する。

【決め事 7】key-value ファイルによる JSON・XML の表現

key-value ファイルの key 列に JSONPath、value 列にその場所の値を割り当てたものを JSONPath-value ファイルとする。
同様に、key 列に XPath を割り当てたものを XPath-value ファイルとする。
（どちらも key-value ファイルの一種である）

2.8.2　変換コマンドと応用例

JSON から JSONPath-value 形式、XML から XPath-value 形式に変換するコマンドはそれぞれ我々が開発済であり、ShellShoccar コマンドセットをインストール済であるならすぐに使える。

例えば、次のような文具の購入明細の JSON データがあったとする。

■bungu_meisai.json

```
{"会員名" : "文具 太郎",
 "購入品" : [ "はさみ",
```

[*8] XML データ中に存在する改行も【決め事 3】に従えば各値を 1 行で表現できる。

[*9] http://goessner.net/articles/JsonPath/

[*10] https://www.w3.org/TR/xpath-31/

```
            "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"}
          ]
}
```

変換コマンドは “parsrj.sh” という名前であり、JSON ファイルを引数または標準入力から与えれば JSONPath-value 形式で出力される。

```
$ parsrj.sh bungu_meisai.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
$
```

ちなみに Twitter API 等、全角文字がエスケープされた状態で来る場合は、“unescj.sh -n”（**-n** オプションを付けた状態）というコマンドをパイプで繋げて実行すればよい。

さて、もし取寄商品の商品名だけ列挙したいという場合は、grep コマンドで絞り込み[*11]、key 列（JSONPath）を除去すればよい。

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

逆変換コマンドは “makrj.sh” という名前である。そこで例えば、「付箋」を追加購入したための元の JSON データを直したい場合を考える。その場合、まずは上記の要領で JSONPath-value ファイルに変換し、付箋に関する一行を追加して、逆変換すればよい。

なお、注意点が 2 つある。一つは、逆変換までする場合には数値や文字列等の方を区別する目的で、parsrj.sh コマンドに “**-t**” オプションを付ける必要があること。もう一つは、購入品配列の添え字（数字）は無視され、JSONPath-value ファイルに書き出した順番で付け直されること[*12]、である。

では、これらの注意点を踏まえて、コマンドの例を例示する。

[*11] もし仮に、商品名（value 列）に「取寄商品」という名前が含まれる可能性があるなら grep は使えないが、代わりに AWK コマンドを使えばよい。

[*12] これは配列要素を自由な位置で追加・削除できるという利点をもたらす。

```
$ parsrj.sh -t bungu_meisai.json >  bungu_meisai.jpv↵
$ echo '$.購入品[0] 付箋' >>  bungu_meisai.jpv↵
$ makrj.sh bungu_meisai.jpv >  bungu_meisai.json↵
$
```

XML の場合は “parsrx.sh” というコマンドを使うことで、XML から XPath-value への変換ができる。なお、XML に関しては、エスケープを解くコマンド・逆変換コマンドは未開発である。

HACK 2.9　CSVindex-value 形式

CSV も、何行何列目にどんな値が入っているのかを列挙すれば、key-value 形式にできる。しかし、行番号・列番号は 2 列で表した方が扱いやすい。

JSON・XML とほぼ同じやり方で CSV ファイルも取り扱い易い形式に変換し、また、逆変換できる。

CSV には JSONPath や XPath のような標準はないが、m 行目 n 列目という 2 つの情報で場所が決まるため、定義は簡単である。例えば、m 行目 n 列目に対しては “/m/n” のように指定すればよい。しかし、長い時間検討を重ねた結果、**“/m/n” のように 1 列に収めるよりも、m を第 1 列、n を第 2 列に分けて管理する方が扱いやすい**と結論づけた。2 列に分かれていれば、例えば「各行の 5 列目以降を抽出する」という指定が、AWK を使って “`awk '$2>=5'`” のように書けるのである。

そこで、key-value 形式から若干アレンジした CSV ファイル取り扱い用の形式を次のように決める。

【決め事 8】CSVindex-value ファイルの定義

元の CSV の各セルの行番号 m と列番号 n を、それぞれ第 1 列、第 2 列に配置するようにして、key-value ファイルフォーマットの第 1 列を拡張したものを CSVindex-value ファイルとする。

2.9.1　変換コマンドと具体例

JSON と同様に、CSV から CSVindex-value への変換とそこからの逆変換コマンドを開発済である。

既に CSV ファイルの例として紹介した次の “自己紹介.csv” を例にとって説明する。

■自己紹介.csv

```
1,井川　さくら,"1995年生まれの秋田出身です。
新参者ですがよろしくお願いします。"
2,千葉　みなと,"1986年生まれ。出身は苗字のとおりです。
去年はだいたい一日16,733人くらいの人と会いました。"
3,武　豊,"明治19年生まれ。出身は愛知県です。
""たけゆたか""ではなくて""たけとよ""です。"
```

変換用のコマンドは “parsrc.sh” である。そして、CSVindex-value ファイルの中身の具体例を次の実行例から確認してもらいたい。

```
$ parsrc.sh 自己紹介.csv↵
1 1 1
1 2 井川 さくら
1 3 1995年生まれの秋田出身です。\n新参者ですがよろしくお願いします。
2 1 2
        :
        :
3 3 明治19年生まれ。出身は愛知県です。\n"たけゆたか"ではなくて"たけとよ"です。
$
```

逆変換用のコマンドは “makrc.sh” である。変換してできた CSVindex-value ファイルをそのままこのコマンドの標準入力に渡せば元通り（等価な CSV）になる。

```
$ parsrc.sh 自己紹介.csv | makrc.sh↵
1,井川 さくら,"1995年生まれの秋田出身です。
新参者ですがよろしくお願いします。"
2,千葉 みなと,"1986年生まれ。出身は苗字のとおりです。
去年はだいたい一日16,733人くらいの人と会いました。"
3,武 豊,"明治19年生まれ。出身は愛知県です。
""たけゆたか""ではなくて""たけとよ""です。"
$
```

HACK 2.10 ファイルは、常にソート済の状態での保存を心がける

整理整頓は、様々な作業を効率化させ、事故防止にも役立つ。File/Dir Hack でも整理整頓を重んじる。データはソート済の状態で保持していれば、JOIN 処理が高速になる。

ここまで、SSV 形式、key-value 形式と、その派生の*-value 形式を取り決めしてきた。この HACK で述べることは「決め事」のように絶対的なルールではないが、これらどの形式にも当てはまり、かつ推奨される話である。

RDB 製品では、そもそもテーブルに保存する際にソートするという概念が無かったので、RDB 製品に慣れ親しんだ人には馴染みづらいかもしれないが、**ファイルに保存すべきデータは、保存するまでの間に、ソートを終えておくべきである。**

各形式での行うべきソートの方針は次のとおり。

ファイル形式	ソート方針
SSV	主キーとして扱いたい列（複数の場合は優先順位をつけて）の文字コード順
key-value	key 列でユーザーが見やすい順にソート
JSONPath-value	【必須】元の順番を維持する。値を追加する場合は、JSON 上で隣り合わせたい値の行同士を隣り合わせる（配列は並べたい順番どおりに）。
XPath-value	JSONPath-value と同じだが、現在のところ逆変換コマンドが無いため、必須ではない。
CSVindex-value	第 1 列（行番号）の数値順 → 第 2 列（列番号）の数値順
（参考）log ファイル	ソートしない。（ログが生成された順番に書き込まれるから、既に日時順にソート済であるといえる）

SSV 形式で、例えば「第 1 列→第 3 列」の優先順で、文字コード順のソートをしたい場合は、次の sort コマンドを通してファイルに保存すればよい。“`-k`” オプションに続いて、ソートさせたい列番号をカンマ区切りで **2 回**書く。

```
sort -b -k 1,1 -k 3,3
```

key-value 形式では、第 1 列のみだからこうなる。

```
sort -b -k 1,1
```

CSVindex-value 形式では、「第 1 列→第 2 列」だが、文字コード順ではなく数値順にしたい。この場合は、1 つ目の列番号の後ろに “`n`” を付ける。

```
sort -b -k 1n,1 -k 2n,2
```

JSONPath-value 形式は sort コマンドでは単純にはソートできない。しかし、parsrj.sh コマンドで生成された行の順番を維持することが基本だ。

2.10.1　何のためにソートするのか?

結論から言うと、対人的には見やすさ、対コンピューター的には join コマンドに掛けるためである。

join コマンドは、RDB 製品における SQL の JOIN 句に相当するものであり、つまり 2 つの表を内部結合または外部結合するためのコマンドだ。内部結合・外部結合は、データ管理では不可欠な操作であるゆえ、join もまた不可欠なコマンドである。

内部結合・外部結合という操作は、ソート済の表に対しては効率のよいアルゴリズムが使えるため、join コマンドは初めからソート（文字列ソート）済のデータしか受けつけないようになっている。（そうでないデータを与えると、データが一部失われたり join コマンドがエラー終了したりする）

join コマンドを使ってみる

join コマンドの使い勝手を知ってもらうため、実際に join コマンドを使ってみよう。そのためには題材となる表を用意しなければならないが、ここでは “`/etc/passwd`” ファイルと、“`/etc/group`” ファイルを用いることにして、どのユーザーがどの第 1 グループに所属しているのかの一覧表を作ってみる。

まずは、group ファイルからグループ名マスターを作る。グループ番号に対するグループ名の対応表ということで、前者を第 1 列、後者を第 2 列にした SSV 形式にすればよい。（HACK 2.7 で補足したように、join する場合には key-value 形式は使えない）なお、group ファイルは各列がコロン記号で区切られているのでそれを半角空白に変換すると同時に、グループ名文字列には半角空白を含んでいる可能性があるので HACK 2.2 の要領でエスケープする。そして、最後にグループ番号（第 1 列）の文字コード順にソートを

掛けて保存する。

```
$ cat /etc/group |↵
> awk -F : '{ gname=$5;↵
>            gsub(/\\/  ,"\022",gname);↵
>            gsub(/_/   ,"\\_" ,gname);↵
>            gsub(/ /   ,"_"   ,gname);↵
>            gsub(/\022/,"\\\\",gname);↵
>            print $1,gname,$3         }' > grpmst.txt↵
$
```

次に、passwd ファイルからユーザー情報ファイルを作る。今回は、1:ユーザー名、2:ユーザー名（詳細）、3:所属する第 1 グループ番号、の 3 列をこの順番で並べることにする。作成方法は、group ファイルとほぼ同じようにやればよい。ただし、こちらは join コマンドの動作確認のために、最後のソート（第 3 列のグループ順）を省いて保存する。

```
$ cat /etc/passwd |↵
> awk -F : '{ uname=$1;↵
>            gsub(/\\/  ,"\022",uname);↵
>            gsub(/_/   ,"\\_" ,uname);↵
>            gsub(/ /   ,"_"   ,uname);↵
>            gsub(/\022/,"\\\\",uname);↵
>            udesc=$5;↵
>            gsub(/\\/  ,"\022",udesc);↵
>            gsub(/_/   ,"\\_" ,udesc);↵
>            gsub(/ /   ,"_"   ,udesc);↵
>            gsub(/\022/,"\\\\",udesc);↵
>            print uname,udesc,$3      }' > uinfo_nosort.txt↵
$
```

最後に両表のグループ番号を結合キーにし、join コマンドで内部結合をする。内部結合に左結合・右結合の区別はないが、我々は**マスターデータを左側に置くという習慣**[*13]にしているので、グループマスターを左表（第 1 表）、ユーザー情報を右表（第 2 表）としている。また、結合したレコードの列構成は、1:ユーザー名（第 2 表第 1 列）、2:ユーザー名（詳細）（第 2 表第 2 列）、3:所属する第 1 グループ名（第 1 表第 2 列）、とする。

[*13] ShellShoccar コマンドセットにも多く移植している Open usp Tukubai の join 系コマンドがそのルールで固定されているため。

```
$ join -1 1 -2 3 -o 2.1,2.2,1.2 grpmst.txt uinfo_nosort.txt↵
     :
（ユーザーの所属グループ表……しかし正しく出力されない）
     :
$
```

すると、表示が途中の行で終了するか、join がエラー終了したはずだ。正しく join させるには、次のようにして、ユーザー情報テーブルを所属する第 1 グループ番号（第 3 列）でソートしてから join する必要がある。

```
$ cat uinfo_nosort.txt |↵
> sort -b -k 3,3       |↵
> join -1 1 -2 3 -o 2.1,2.2,1.2 grpmst.txt - |↵
> sort -k 1,1          |↵    ←見やすさのため、ユーザー名でソートした
> keta --↵                  ←見やすさのため、さらに列揃えの keta コマンドを通した
    :
（ユーザーの所属グループ表……今度は正しく出力される）
    :
$
```

このようにして、RDB 的操作で必要不可欠な結合操作が、、File/Dir Hack でもできるのである。

ちなみに ShellShoccar コマンドセットには、Open usp Tukubai から移植した “cjoin1”、“cjoin2”、“cjoin0” というコマンド[*14]があって、これらではソートされていない右表を受け入れられる。通常の join と比べて一長一短がある[*15]のだが、その解説は本書の今後の改訂版で追加する予定である。

HACK 2.11　一時ファイルも必要なら使う

一時ファイルは使わないに越したことはないが、必要ならば気にせず使う。後始末を面倒に感じて避けるあまり、複雑なプログラムにしてしまう方が後々面倒。

2.10.1 項で join コマンドが出てきたが、このコマンドを使うには join する双方のデータがソート済という前提条件があった。もしどちらのデータ（ファイル）も未ソートであったら、少なくともどちらかのデータは予めソートを済ませおかなければならない。なぜなら、一連の処理の流れの中でソートを行おうとしたら標準入出力を使わなければならないが、1 つのコマンドが受け取れる標準入力は 1 つだけだからである。確かに UNIX には「名前付きパイプ」という機構があり、これを使えば 1 つのコマンドに複数の標準出力を送れると言えなくもないが、今問題にしているソートという処理においては、一時ファイルを使う以上のメリットはさほどない（後述）。このような理由で一時ファイルを使うのが適当と判断されるのであれば、一時ファイルを使うことを過度に避けるべきではない。

[*14] `https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_cjoin1` 他

[*15] 左表のマスターデータをすべてメモリ上に読み込んでから結合する仕組み。そのため列数が少ない場合（数百行程度まで）は有効だが、多すぎるとメモリ不足に陥る。

一時ファイルを避けたい理由は、

a. パイプに比べて遥かに時間のかかるファイル I/O が発生すること
b. 一時ファイルが他と衝突しないように固有の名前をつける必要があること
c. 一時ファイルを他ユーザーに破壊されないように気を付ける必要があること
d. 一時ファイルが消されず残ってしまう可能性があり、ディスク上のゴミと化すこと

などがある。だからといっていつでも極力避ける方がよいとも限らない。ソート処理などはその典型例だ。ソートはすべてのデータを読み込まないと処理が確定しないので、join に必要な 2 つのデータを（名前付きパイプを使うなどにより）同時にオンメモリでやろうとすれば、両方のデータの最初から最後までのすべてがメモリを消費する瞬間がある。ファイル I/O の発生は抑えられるが、データが巨大だと、代わりにメモリの大量消費が問題になり得る。また最後まで読み込まないと処理が確定しないがゆえに、名前付きパイプを使うメリットであるストリーミング（到来した行から次々と処理を完了させていく）効果も得られないどころか、一時ファイルと同様に b〜d の問題だけが残る[*16]。

b〜d の問題は一時ファイルと名前付きパイプのどちらにも付きまとうが、これらに関しては次のようなスニペットをシェルスクリプトに書き込むことで大幅に低減できる。

2.11.1 一時ファイルをより安全に作成・削除するスニペット

一時ファイルの作成において、衝突を避け、他人の攻撃（あるいは誤操作）から守り、そして極力ゴミとして残らないようにするために、一時ファイルを使うシェルスクリプトには次のようなスニペットを書くことをお勧めする。

■冒頭に置くスニペット

```
# シェルスクリプトの正常・異常終了(SIGKILL以外)で呼ばれるシェル関数の定義
exit_trap() {
  set -- ${1:-} $?  # $? is set as $1 if no argument given
  trap '' EXIT HUP INT QUIT PIPE ALRM TERM
  [ -d "${Tmp:-}" ] && rm -rf "${Tmp%/*}/_${Tmp##*/_}"
  trap - EXIT HUP INT QUIT PIPE ALRM TERM
  exit $1
}

# 上記シェル関数を有効化（トラップを定義）し、一時ファイル用ディレクトリーを作る
# * このコードは、必ずしも冒頭に書かなくてよく、最初の一時ファイルを作る必要が
#   生じる箇所の直前に書いてもよい。（ただしサブシェルの中に書いてはいけない）
trap 'exit_trap' EXIT HUP INT QUIT PIPE ALRM TERM
Tmp=`mktemp -d -t "_${0##*/}.$$.XXXXXXXXXX"` || {
  echo "${0##*/}: Failed to mktemp" 1>&2
  exit 1
}
```

[*16] bash 等の一部のシェルでは「プロセス置換」を使うことで名前付きパイプを管理せずに済むようにできるが、一部のシェルでしかできないことは POSIX 原理主義に反する。

このスニペットについて解説する。

仕組みは、trap コマンドを利用し、正常または異常終了で何らかのシグナルが到来した場合に、一時ファイル用のディレクトリー（存在すれば）を消すためのシェル関数を呼び出すというものである（SIGKILL は捕捉できないのでこれが来た場合にはどうしようもないが）。捕捉可能なシグナルのうち、ここでは HUP、INT、QUIT、PIPE、ALRM、TERM の 6 種類しか記述していないが、他のシグナルが到来する可能性があるプログラムの中では適宜追加して使う。

mktemp コマンドで、一時ファイルではなく一時ディレクトリーを作っている。こうすれば、後述のようなコードにより、そのディレクトリーの中に後から好きなだけ一時ファイルを作れる。また、mktemp でディレクトリーを作るとパーミッションコードは 700 になるので他ユーザーの誤操作や攻撃、盗聴から守れる。mktemp コマンドに “`-t`” オプションが付けてあったり、このような名前にしているのは、様々な環境にある mktemp コマンド（少しずつ挙動が違う）で名前衝突を避けるための経験則に基づいている。こうして作成された一時ファイル用ディレクトリーパスは、シェル関数 “`$Tmp`” に入る。

なお、mktemp コマンドは POSIX で規定されたものではないため存在しない環境もあるが、HACK 1.7 で紹介した ShellShoccar コマンドセットの中には POSIX 移植版 mktemp コマンドも用意してあるのでこちらにパスを通せば POSIX 準拠が担保できる。

■一時ファイルの作り方・使い方の例

```
sort -bk 1,1 unsorted_masterfile.txt > "$Tmp/sorted_masterfile"

: > "$Tmp/result"
if [ -s "$Tmp/sorted_masterfile"]; then
  cat unsorted_transactionfile.txt          |
  sort -bk 1,1                              |
  join -1 1 -2 1 "$Tmp/sorted_masterfile" - > "$Tmp/result"
fi

echo "$(cat "$Tmp/result" | wc -l) line(s) joined."
```

上記の例では、2 つの一時ファイルを作っている。実際に一時ファイルを使いたい時に改めて mktemp コマンド実行する必要はなく、“`$Tmp/`” の後ろに好きな名前を書くだけでよい。ただし mktemp コマンドと違い、それを書いただけでファイルができているわけではないので、いきなり読み込もうとすれば “No such file or directory” エラーになってしまう。上記の例では、そうなってしまわないよう “`$Tmp/result`” については先に空の一時ファイルを作っている。

第 3 章

データ管理のためのディレクトリーの使い方

RDB 製品を用いず、ファイル・ディレクトリーを用いたデータ管理をする際の強力な仕組み。それはまさに、ディレクトリーである。強力な仕組みであるということは、**ディレクトリーの使い方を知らなければデータ管理においては無力**という意味の裏返しでもある。

HACK 3.1　RDB のテーブル 1 つ ≠ ファイル 1 つ

テーブルとファイルはそれぞれ、できる事とできない事に差異があり、そのまま置き換えはできない。だいたい、ファイルの方がデータをより細かく持つことになる。

ファイルとディレクトリーでデータ管理をしろと言われたら、多くの人はまず、使い慣れている RDB 製品の概念（テーブルや SQL コマンド）がどう対応するかを考えるだろう。すると、データを格納する容器の基本単位は、RDB 製品ではテーブルで、ファイル・ディレクトリーの場合にはファイルであるから、各テーブルの 1 つずつを、そのまま 1 つずつのファイルで管理すればよいと考えるかもしれない。

しかし、**RDB 製品のテーブルと File/Dir Hack におけるファイルは、一対一では対応しない。**理由は、両者の取り扱い上の性質に違いがあり過ぎるためである。

次の表を見てもらいたい。テーブルやファイルに対する操作で、（一般的に）[*1]可能・不可能が異なりそうなものをいくつか列挙してみた。一番右の列には参考に、ディレクトリーについての可否を記した。

[*1] 製品によっては可能・不可能が異なるかもしれないが、ここでは考えない。

操作	RDB の table	File	Dir
書込時のソート	不要	推奨	不要
末尾へのレコード追加	可	可	可
途中位置のレコード更新・削除	可	不可	可
(中身でなく) 自身の階層化	困難	Dir 併用で可	可
排他ロック（全行）	可	可	不可
排他ロック（各行）	可	不可	不可
共有ロック（全行）	可	不可	不可
共有ロック（各行）	可	不可	不可

パッと思いついただけでもこれだけの違いがある。

3.1.1 ランダムアクセスのテーブル、シーケンシャルアクセスのファイル

最初の 3 つは、テーブルとファイル、それぞれがランダムアクセスかシーケンシャルアクセスかの違いに起因している。テーブルはランダムアクセス方式であるため、途中の位置でのレコード更新・削除が自由にできる[*2]のに対し、ファイルはシーケンシャルアクセス方式であるため、それができない。

書き込み時のソートというのも、シーケンシャルアクセスならではの事情だ。ファイルに格納する際には、目的の列に対するソートを終えた状態にしておくことが推奨される。そうしなければ、ファイルを最初から最後までを読み終えてみないと、目的のレコードを探し出すことが難しくなるからだ。そういう性質もあって、2 つの表を内部・外部結合する join コマンドも、初めからソート済であるデータしか受け付けない[*3]。

これがファイルの弱点ではあるが、その代わり、ディレクトリーがランダムアクセスなので、1 つのファイルだったものを、種類別に複数のファイルやディレクトリーに分けて管理することでこの弱点を克服できる。

3.1.2 ファイルなら、階層化して格納できる

テーブルは、ディレクトリーに相当するものを使ってその中に整頓するということが難しかったのに対し、ファイルはディクトリーを駆使することで階層化して格納できる[*4]ことが利点である。よって、ファイルはテーブルよりも小さく分割して管理しやすい。

この性質は、ファイルでは不可能な行単位のロックに近いことを実現するのにも役立つ。条件に応じてファイルを細かくすれば、そのぶん細かくロックができるからである。

これらの考察から、テーブルとファイルは一対一に対応させるべきものではないことが理解できるだろう。ここでは一対一ではないことを頭に入れておけばいい。どう対応させるべきかはケースバイケースなので、また別の HACK で解説する。

[*2] 厳密には、SELECT 文で出力されるまで、レコードの順番が定まらない。

[*3] 一般的に、SQL の SELECT 文に出てくる JOIN 句も、内部的にはソートを行っている。

[*4] データの中身の階層化ではなく、データが格納されているテーブルやファイル自身を階層管理できるという議論であることに注意。

HACK 3.2　1 つのディレクトリー直下のファイル数 ≦10000

多くの UNIX 系 OS では、シェルやコマンド内部でソートや重複チェック等が走り、ファイル数に比例以上の計算リソースを消費するので少ない方がいい。経験上、上限は 10000 程度にすべきだ。

HACK 3.1 にて、「ファイルはシーケンシャルアクセスだから、ランダムアクセス的なことがしたければ、レコードを種類に応じて複数のファイルに分割した方がよい」という旨の説明をした。ならば究極的には、1 レコード 1 ファイルにすればよいと思うかもしれないが、何事も物には限度というものがある。仮に 100 万レコードから成るデータがあったとして、それらの各レコードを単独で 1 つのファイルにしたならば、処理が遅くて使い物にならないだろう。

3.2.1　同一ディレクトリーに大量のファイルを作る実験

論より証拠。あなたの UNIX 環境で、次のコードを試してみてもらいたい。これは、“files” というディレクトリーの中に、変数 n で設定した数だけファイルを生成するためのコマンドだ。変数 n は、“10” の場合、“100” の場合、……、と 1 桁ずつ増やしながら試していき、“1000000”（100 万）くらいまで試してもらいたい。

```
$ n=ここに任意の整数↵
$ rm -rf files && mkdir files && (cd files && time seq 1 $n | xargs
touch)\return
```

こちらの環境（CPU:Corei5-6500、HDD:WD30EFRX、OS:FreeBSD11.1R）で試した結果は次のとおりである。

n の値	realtime（所要時間）
10	0.01 秒未満
100	0.01 秒未満
1,000	0.01 秒未満
10,000	0.01 秒未満
100,000	0.37 秒
1,000,000	3440.96 秒

n が 1 万までの時は、測定限界以下（0.01 秒未満）だったものの、10 倍の 10 万になると 0.37 秒と、10 倍になっただけなのに少なく見積もっても 37 倍となり、さらに 10 倍の 100 万になると、10 倍になっただけなのに 9000 倍以上の時間が掛かっている。

ここでは実験していないが、上記の方法で “files” ディレクトリーの中に大量のファイルができあがった後、ls コマンドで files ディレクトリー内のファイル一覧を取得すべく ls コマンドを実行してみれば、そちらもファイル数が増えるほどに劇的に遅くなることがわかる。

3.2.2 考察

ファイルを新規作成する際は、一意制約確認（同一ディレクトリー内に重複したファイル名は存在できないため）とインデックスの生成（作成されたファイルをすぐに見つけられるように）が内部で毎回行われる。ファイル数が多ければ多いほどこれらの処理には時間がかかるわけで、これがファイル数に比例する以上の時間を要する原因である。

ちなみに、ls コマンドやワイルドカード指定等でファイル名を表示する際は、コマンドやシェル内部でファイル名のソート処理が走る。ソートは一般的に、ソート対象の個数に比例以上の時間を要するうえ、我々の見立てでは、ls コマンドやシェルのファイル名ソートアルゴリズムはあまり賢くないと思っている。

これらのことから推察するに、**UNIX 的には、1 つのディレクトリーに大量のファイルを置く使われ方を想定していないのではないか**と思う。ではどのくらいの数までならいいのかという議論になるが、実験結果からは 10000 ファイル程度が限界だろうと言える。そして、我々のこれまでの経験からも、これは妥当な数だと考えている。

HACK 3.3 内容更新する 1 つのファイルに収める行数 ≦10000

大量レコードから成るデータがある時、1 つの大きなファイルで管理するのも、大量の細かなファイルに分けて管理するのも一長一短がある。どの程度に分けるのが適切なのか、実験し、検証した。

3.3.1 大きい 1 つのファイルと細かい大量のファイル

既に述べたとおり、ファイルは RDB 製品のようにランダムアクセス的なことができない。途中に行を追加、あるいは削除したり、修正するなどしたければ、ファイル全体を読みながら編集を施して別ファイルに書き出し、それを元ファイルに上書きするという操作をしなければならない。ということは、大量のレコードから成る巨大ファイルほど更新に時間がかかってしまう。

一方、そういう無駄を避けたいと思って、レコードをいくつものグループに分割し、複数のファイルに分けて管理すれば、該当するレコードを 1 つの小さなファイルだけ更新すれば良いので更新は早くなる。しかし、全レコードを対象とした検索を実行しなければならない場合、分割した全ファイルを順次オープンする必要が生じる。ファイルのオープン処理は意外と時間が掛かるため、この場合は細かく分割するほど時間がかかると予想される。

つまり、

細かくすると…… ファイルをオープン・クローズする時間が増える
大きくすると…… ファイルに書き直す時間（サイズ）が増える

という問題が起こる。ならば、1 ファイルあたりのレコード数はどれくらいを超えたら分割すべきなのか?

コラム

一時ファイルを作らないファイル更新テクニック 2 つ

UNIX 系 OS 上では一般的に、ファイルの内容を逐次読み込み、逐次編集し、そして自分自身に逐次書き込みをしてはいけない（壊れてしまう）。一見、それができているように見える場合でも、内部で一時ファイルを経由させていたり（sed の “-i” オプション）、一旦メモリに全内容を読み出してから書き戻す（vi コマンド等）ようにしている。しかしながら、一時ファイルを介するのは（消し忘れるとゴミになる等で）面倒なので、できれば避けたい。そこで、自分で一時ファイルを作らずにファイルの更新を行うテクニックを 2 つ紹介する。

■同じ名前の別ファイルを作る　1 つ目の方法は、ファイルを開くと同時に削除して、同名のファイルを新規作成するというトリッキーな方法である。例えば、hoge.txt というファイルの中にある小文字をすべて大文字に、一時ファイルを介さず行うには次のようにする。

```
(rm hoge.txt && tr a-z A-Z > hoge.txt) < hoge.txt
```

UNIX 系 OS では、オープンしている最中のファイルを削除しても、クローズするまではファイルディスクリプター（ハンドル）が有効なので読み続けられるが、既に名前は失っているので、元と同じ名前（ただし実体は別）のファイルを作成できるため、この性質を利用している。このテクニックは、tr コマンドのみならず、パイプで入出力できるどんなコマンドにも使えるし、パイプを使って複数のコマンドをつなげることもできる。
ただし、実体は別ファイルなので、ファイルパーミッションや所有者、ACL 等の情報は失われ、自分で新規作成した時の状態になってしまうので注意。

■greo コマンドを使う　2 つ目は、ShellSchoccar コマンドセットに収録されている “greo”（Global Regular Expression and **Overwrite**）というコマンドを使う方法だ。greo コマンドは、grep コマンドのような使い心地で、その名のとおりグローバル置換を行ってくれるコマンドであり、そしてコマンド内部で一時ファイルを作成・始末してくれるのでユーザーは意識する必要がない。
例えば、管理している web サイトにある HTML ファイルのリンクを一斉に更新したいなら次のように書けばよい。

```
find ~/hogeWebDir -name '*.html' | xargs greo -Fp 'old-URL' 'new-URL'
```

“-F” オプションは、grep のそれと同様、正規表現を無視するもの（-F 無しの場合や-E 付きの場合は正規表現と見なす）で、-p オプションは本番（グローバル置換は取り消しが効かないので-p 無しだとリハーサルモードになる）の意味である。詳しくは “--help” オプション参照。

いずれもファイルの更新には便利なテクニックなので、是非活用してみてもらいたい。

3.3.2　1000 万レコードファイルの持ち方実験

手元に、頻繁に内容が更新される 1000 万レコード（行）に及ぶデータがあったと想定し、これを何ファイルに分割（1 ファイルあたり何行くらいに）するのが適切なのかを調べるため、次の実験をした。

まず、次のシェルスクリプトを実行して検証用のダミーデータを生成した。

```
#!/bin/sh
total=1000000
for lines in 10 100 1000 10000 100000 1000000; do
```

```
  files=$((total/lines))
  echo "($files)"
  dir="lines-$lines"
  mkdir $dir
  i=1
  while [ $i -le $files ]; do
    od -t x4 /dev/urandom |
    sed 's/  */ /g'       |
    sed -e "N;N;s/\n//g"  |
    head -n $lines        > "$dir/no$i"
    i=$((i+1))
  done
done
```

これで、1 レコード 129 バイト[*5]で 1000 万レコードが生成されるが、次の 6 通りのパターンで生成される。

- 1 ファイル 10 レコード * 1000000 個
- 1 ファイル 100 レコード * 100000 個
- 1 ファイル 1000 レコード * 10000 個
- 1 ファイル 10000 レコード * 1000 個
- 1 ファイル 100000 レコード * 100 個
- 1 ファイル 1000000 レコード * 10 個
- 1 ファイル 10000000 レコード * 1 個

そして、各パターンのファイル群に対して、（全検索を想定した）全ファイルリードと、（部分更新を想定した）1 ファイルコピーの所要時間を計測した。

```
■全ファイルリード
echo * | xargs cat > /dev/null

■1ファイルコピー
cp no1 no1-1
```

先程と同じ環境（CPU:Corei5-6500、HDD:WD30EFRX、OS:FreeBSD11.1R）で試した結果は次のとおりである。

ファイル構成	全ファイル読み	1 ファイルコピー
100 行 * 100000 個	19.71 秒	**0.01 秒**
1000 行 * 10000 個	**14.98 秒**	**0.01 秒**
10000 行 * 1000 個	**14.75 秒**	**0.03 秒**
100000 行 * 100 個	**13.82 秒**	0.28 秒
1000000 行 * 10 個	**14.04 秒**	2.65 秒
10000000 行 * 1 個	**10.27 秒**	23.65 秒

[*5] 1 つのレコードをあまり横に伸ばすのはよくない。だいたい 120 バイト程度までが理想と考えた。

全ファイル読みは、ファイル個数 1 万以下（1 ファイル行数 1000 行以上）になるとだいたい落ち着いている。一方、1 ファイルコピーは、1 ファイル 1 万行以上（ファイル個数 1000 個以下）では所要時間がほぼ行数に比例しているが、それ以下では他の要因の影響を受けて違いがなくなっている。

3.3.3　1 ファイルに持たせるべきレコード数の上限は?

この実験に基づけば、頻繁に更新が発生するデータに関しては、**1 ファイルのレコード数が 1000〜10000 程度に達したら、分割を検討するのがよい**ということになる。そして、これは我々の経験とも一致する。

ただし、個々の条件によっても臨機応変に判断するべきだ。上記の実験結果では分割を検討すべきレコード数は 1000〜10000 と幅があるが、1 レコードのサイズが小さいなら 10000 のオーダーでもいいだろうし、逆に大きいなら 1000 のオーダーでも検討すべきだろう。それ以外の判断材料としては、全検索を頻繁に行わないならもっと細かくしてもよいだろうし、逆に更新頻度がそれほどでもないというならレコード数を増やしてもよいだろう。とにかく、取り扱うデータの性質を見ながら判断すべきだ。

3.3.4　会員データはどう管理するか

大量レコードを持つデータの分割管理を議論する際の典型例は会員データである。中小企業のように数十〜数百名程度の小さな会員データあれば、100 万人を超えるような大規模 web サービスの会員データもあり、スケールの幅が広いからだ。

ここまでの HACK を鑑みれば、レコード数（つまり会員数）は 10000 程度に達する場合に分割すべきということになる。分割の仕方は、例えば会員の本名ふりがなの 50 音順に「あ行」、「か行」、……、「わ行」のように 10 分割するアイデアもある。それで不十分と見込まれるならば、「あ」、「い」、……のように 50 分割にするのも考え方だ。

例えば会員 ID を主キーとして扱っており、そちらの順番に並べて管理する方が都合がよいケースなら、会員 ID の先頭英数字で分割するという手もある。もしそれで不十分なら 2 文字目も見ればいい。

ただ、いずれの分け方の場合も分布に偏りが生じ得るので、1 ファイルのレコード数はある程度余裕を持たせておいた方がよいかもしれない。

更新するとソート処理が発生するが、その考慮は?

例えば、「ま行」を管理している会員データファイルで、……、「まつうら」さん、「まつしま」さん、「まつやま」さん、……と続くデータにがあったとして、

■members_MA.txt

```
   :
まつうら matsuura@nagasaki.example.jp
まつしま matsushima@miyagi.example.jp
まつやま matsuyama@ehime.example.jp
   :
```

「まつお」さんのレコードを挿入する場合を考える。

ふりがなでソートされた状態を維持しようと思ったら、まつおさんのレコードを一旦最終行の次に書き足して、ソートをしなければならないと思うかもしれない。しかしながらその方法だと、レコード数が多くな

るほどソートにかかる計算量が急増し、非常に効率が悪くなってしまう。

でも心配には及ばない。マージソートというソートアルゴリズムを使えば計算量は遥かに低く抑えられる。幸いにも sort コマンドには “-m” オプションでこれが実装されている。マージソートをする場合には、追加したいレコードをファイルに書き足してから sort コマンドに渡すのではなく、次のように結合したいデータ同士を引数で並べる（3 つ以上も可）。

```
$ echo 'まつお matsuo@mie.example.jp' |↵
> sort -bm -k 1,1 members_MA.txt -    > members_MA.txt.tmp &&↵
> mv members_MA.txt.tmp members_MA.txt↵
$
```

ShellShoccar コマンドセットには、Open usp Tukubai から移植した “up3” というコマンド[*6]があってさらに使いやすいのだが、その解説は 7.2.2 の中の「ソート順を維持しながら追記する（安定ソートが必要な場合）」項に記してある。

[*6] https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_up3

第 4 章

実は排他制御不要な排他処理

複数のプログラムやユーザーからのアクセスを受け付けるシステムを設計すると、トランザクション処理、つまりそれをやり始めたら邪魔されることなく最後までやり終えなければならない処理が大抵生じる。そんな時、他のプロセスの侵入を防ぐための排他制御を行うが、よくよく考えれば不要なものもある。

File/Dir Hack ではそういう場面が多いため、それらのケースを知っておくことはとても重要である。

HACK 4.1　単調増加カウンターは追記リダイレクションで作る

カウンター更新時のファイルロック不要。ディスクドライブが同時に 1 つのリクエストしか受け付けられない必然性を利用する。

例えば Web サイトのアクセスカウンターを作ることを考える。全 Web ページ共通のヘッダー部分に CGI スクリプトを仕込んでおき、それがアクセスされるたびにカウンターをインクリメント（+1）する。では、そのカウンターは具体的に、どういうデータ構造にすべきか。

> どうもこうも、一つのファイルに現在値をテキストで書き留めるに決まっているじゃないか。それ以外にどうしろと?

いや、我らはそうはしない。

4.1.1　ファイルサイズをカウンターとして使う

先にアイデアを言ってしまうと、カウンター値をファイルの中に書くのではなく、ファイルサイズをカウンター代わりに使う。例えば、ファイルサイズが 1000 バイトであったらこれまでに 1000 アクセスあったと見なす。1000 バイトのデータは具体的に何にするのかと気になるかもしれないが、何でもよい。任意の半角文字でも、改行文字でも。

以下はカウンターとするファイルが “*counter.dat*” という名前であるという前提で話をする。

カウンターインクリメント

1 回アクセスがあったら、

```
echo '' >> counter.dat
```

などとやれはよい。

もし何かシステムの都合で一度に 100 のインクリメントをしたいのなら、echo コマンドを 100 回実行するのも効率が悪いので少し頭を捻る。たとえば、printf コマンドを使えばよいだろう。

```
n=100        （変数 n の値を変えればインクリメント数をいくつにでもできる）
printf "%0${n}d" 0 >> counter.dat
```

printf コマンドは、"`%03d`" などと書けば 3 桁（つまり 3 文字）の文字列が生成されるのでこの性質を利用するのである。

なお、どちらの方法も、カウンターを更新する際にファイルロックをしなくてよい。（Web アクセスカウンター等の場合は）

カウンター参照

一方、カウンターの値を読みだす場合はどうするのかといえば、次のようにして ls コマンドで簡単に調べられる。

```
ls -l counter.dat | awk '{print $5}'
```

"`-l`" オプションを付けて詳細表示にすれば 5 列目にサイズが返されるので、それを AWK で取り出せばよい。

なお、この時も、カウンターを参照する際にファイルロックをしなくてよい。（Web アクセスカウンター等の場合は）

4.1.2 なぜファイルロック不要なのか

後の節で説明するが、POSIX にはファイルをロックしてくれるコマンドがないため、ファイルロック（に相当する処理）を実現するのは結構手間だ。だからロック不要なアイデアを捻り出すことが重要である。

そもそも通常のカウンターを更新する際、なぜロックが必要になるのだろうか。それは、更新する前に現在値を一度読み出して次の値を計算し、それを書き戻すという複数の処理を経なければならず、なおかつその間には他の誰にも邪魔されてはならないからである。ということは仮に、処理が一つであれば邪魔される隙が原理的になくなるのでロックの必要がなくなるということである。（これを**アトミックな処理**という）

そんな都合の良い処理が……、よーく見まわすと追記リダイレクションにあった。

ブロックサイズ以下のファイル追記はアトミックである

UNIX においては、1 ブロック分のサイズ以下のデータをファイルに追記する場合、アトミックに行われる。1 ブロックのサイズはというのは UNIX においては 4096 バイト以上であるから、すなわち **4096 バイト以下のデータをファイル追記すると、そのデータは必ず連続した領域に書き込まれる。**

これはどうやっても邪魔できない。仮にあなた達が別々にターミナルを起動して、それぞれ別の 4096 文字以下の文字列を同じファイルに「いっせーの」で追記しても、絶対に両者の文字列は 1 文字も失われない

し、混ざりもしない（どちらかの文字列が相手の文字列の途中に割り込まない）。カウンターという用途においては、1 文字も失われないという性質に目を付け、利用している。

そもそもなぜファイル追記はアトミックなのかといえば、それはディスク装置というハードウェアが、瞬間的には一つのリードまたはライトしか受け付けられないからである。一方で UNIX 系 OS 自体はマルチタスクで動いており、この矛盾を解決しなければならない。だから OS はカーネル上で、必ずディスクアクセスの排他制御とキューイングを行っているはずである。

このようにして、OS 自体が本質的に持っている性質まで利用し尽すのが File/Dir Hack の醍醐味である。

4.1.3　デメリットと限界

ただ、この方式も万能とは言えない。弱点もある。

ディスク領域の無駄遣い問題

ファイルにカウンター値のテキストを持たせる場合なら、ファイルサイズは桁数分のバイト数で済む。しかし、ファイルサイズを値とする場合にはディスク領域が大量に消費される。

しかし、費用対効果を熟慮して考えるべきだ。仮に 100 万までカウントした場合、消費されるディスク領域は約 1MB である。更にその 1000 倍の 10 億までカウントしたとしても約 1GB だ。TB オーダーのディスクがポケットマネーで買える時代、さほど気にする必要はないのではないか。POSIX 原理主義の恩恵（高い互換性と長い寿命）と天秤にかけながら判断してもらいたい。

デクリメント（−1）するにはさらに一工夫

例えば、ある施設に滞在中の人数を管理したい場合を考える。入場する人もいれば退場する人もいるために、この場合には増えるのみならず減りもする。こういう目的に対しては通用しない。

ただし、入場者カウンターと退場者カウンターという 2 つのファイルを用意してそれぞれの増加をこの仕組みで記録し、参照したい時には前者から後者を引くというアイデアを駆使すれば実現可能だ。

カウントした直後の番号を正確に知りたい場合はロックが必要

ファイル追記という操作は、ある意味箱にボールを投入する操作と同じである。今箱にボールが何個入っているかを調べることなく追加できるという意味においてだ。ところが、「今自分がボールを入れた結果、箱の中のボールの数はいくつになったのだろうか」が知りたいという場合は、数える必要が生じてしまう。数えている間に別の誰かがボールを入れたらわからなくなってしまうので、数え終わるまで入れさせてはならない。つまり、ロックが必要だ。

Web アクセスカウンターの場合は、そこまでの正確さは必ずしも求められないのでこれでもよかった。しかし、この章の目的である受付番号を発行するという場合にはそうはいかないため、ロックが必要になってしまう。ロック（に相当する）操作を実現する方法は、5 章の HACK で紹介する。

HACK 4.2 複数プロセスから同一ファイルの内容を更新しない設計にする

更新するのが一人なら、そもそも競合する相手がいないから絶対安全。

先程の HACK のように単調増加ではないカウンターとして、例えば数ある商品の在庫数を管理することを考えてみる。単調増加ではないからもはや先程の HACK のようにファイルサイズで管理する方法は有効ではない。そこで、商品 ID（例えば JAN コード）と在庫数という 2 列で構成されるテキストファイルで管理することにした。

■在庫数管理ファイル “stock_qty.txt”（1 列目:JAN コード、2 列目:在庫数）

```
4930726001394 55
4930726100219 151
4930726100233 74
      :       :
      :       :
```

在庫数は変わっていくから、それに応じてこのファイルも更新しなければならない。この時もし、在庫数を書き換えるプログラムが複数存在したり、あるいは同じプログラムが同時に複数起動して書き換えに来る可能性があるなら、排他制御が必要になってしまう。そうしなければ、最悪の場合にデータが壊れるからだ。しかし、在庫帳の書き換え役が一人であるような設計にできれば、データは壊れなくなる。書き換えの邪魔をする相手がいないのだから自明だ。

これなら排他制御は必要なくなる。例えば、在庫数の書き換えは、定期的に起動するバッチスクリプトが担当するようにしたり、書き換えを希望するプロセスからの要求に応じはするものの別プロセスとして非同期に立ち上がり、そのプロセスの都合で書き換えるようにするなどのやり方だ。

ただ、読み取りプロセスが別に存在する場合にはこれでは不十分なので、もう一工夫必要だ。それについては次の HACK で説明する。

HACK 4.3 ファイル更新の最後には mv コマンドを使う

同一パーティション内の mv はアトミックに行われるので、書き換え途中のファイルを見せずに済む。

前の HACK を読んでいて、

> それなら確かにデータは壊れないが、他に在庫数を読み出したいプロセスがて、書き換え中に読みに来たらどうするのか?
> 例えば 1000 種類ある全商品の在庫数書き直し処理を行っていて、もし 500 番目までしか書き換え終えていないタイミングで、別プロセスが 501 番目を読み込もうとしたら、レコードが無いといってエラーになってしまう。

という疑問を抱いた人もいるだろう。その懸念は正しい。この問題を防ぐためには、元ファイルの更新に mv コマンドを使わなくてはならない。

4.3.1　良い例と悪い例

前の例の続きで、在庫数を更新することを考えるとしよう。今、在庫数ファイルが‘‘$DirData’’というディレクトリーの中の“stock_qty.txt”というファイルに格納されていて、最新の在庫数を返すシェルスクリプト“latest_stockqty.sh”を使って更新をしようとしている。在庫データの書き換え途中を見せてしまって事故を起こさないようにするための良い例と悪い例をいくつか示す。

■書き換え途中のファイルを読まれる事故を防ぐため対策

```
○悪い例 1（書き換え中の不完全なデータを読み取られる可能性がある）
latest_stockqty.sh > "$DirData/stock_qty.txt"

○悪い例 2（これも同じ）
latest_stockqty.sh > "/tmp/stock_qty.txt"         &&
cp "/tmp/stock_qty.txt" "$DirData/stock_qty.txt"

○良い例（mv によるファイルの差し替えは一瞬で終わる）
latest_stockqty.sh > "$DirData/.stock_qty.txt.tmp"         &&
mv "$DirData/.stock_qty.txt.tmp" "$DirData/stock_qty.txt"

○あまり良くない例（もし移動元と先でパーティションが違うと一瞬で終わらない）
latest_stockqty.sh > "/tmp/stock_qty.txt.tmp"         &&
mv "/tmp/stock_qty.txt.tmp" "$DirData/stock_qty.txt"
```

悪い例 1、これは皆で開くファイルである“latest_stockqty.txt”を直接編集してしまっているので一番事故に遭いやすい。悪い例 2 では一時ファイルを経由して cp コマンドで元ファイルに上書き更新しているが、コピー時にはデータ転送時間が発生するので完璧とは言えない。正しくは次の「良い例」のようにして、同じディレクトリーに隠しファイルなどの形でファイルを一旦作り、完了後に mv コマンドで差し替える。差し替え処理にはデータ転送が無いのでアトミックに（一瞬で）終えられる。

注意が必要なのは、その次の例である。一時ファイルはそれ用のディレクトリーに作るべきと考え、“/tmp”ディレクトリーで作成した後、mv コマンドを使って元のファイルに上書きしている。しかし、もしも“/tmp”と移動先のパーティションが違う場合は cp コマンドと同様にデータ転送が発生してしまうので同じディレクトリー作る方が確実だ[*1]。

HACK 4.4　小さなファイルをディレクトリーに並べる

ディレクトリーはランダムアクセス。ファイルを小さく分割して並べれば、ランダムアクセス性が高まる。

これは先の二つの HACK の応用、あるいはそれらと組み合わせて使うと有効（な場合がある）、というものである。

[*1] SELinux を用いている場合、“/tmp”に作ったファイルを移動や複製すると別の問題も招く。“/tmp”で新規作成した時に付けられた一時ファイルとしてのコンテキストも一緒に移動するため、不具合が生じてしまう。

4.4.1 在庫数ファイルによる例

例えば先程の例では、1000 種類ある商品の在庫数を一つのファイルで管理していた。HACK 3.2 の経験則に基づけば、一つのディレクトリーに 1 万個程度までのファイルを置いてもさほどパフォーマンスに影響しないので、商品 1 種類につき 1 ファイルで管理してみるのもいいかもしれない。

つまり、先程の在庫数ファイル “stock_qty.txt” を次のようにして 1 商品 1 ファイル化しておく。

```
$ mkdir stock_qtys↵
$ awk '{print $0 > "stock_qtys/" $1;}' stock_qty.txt↵
```

これで、「1 列目:JAN コード 2 列目:在庫数」という列構成で 1 行で構成されたファイルが、各々 1 列目（JAN コード）のファイル名で生成される。1 行目の JAN コードはファイル名にあるのでファイル内に書く必要はないと考えるならば、上記のコードで “$0” と記述されている箇所を “$2” にすればいい。

では、こうすることの何がいいのか。一つのファイルで管理していた時の問題点を考えてみればわかる。

- 読み込む際は、必要な行のみならず、少なくとも必要な行に到達するまで読み出さなければならず、時間も掛かるし効率も悪い。
- 書き換えをする際も、毎回すべての行を読み出して書き直さなければならず、やはり非効率である。
- もし書き換え時に排他制御が必要だったとすると、在庫数データ全体を対象としなければならない。

つまり、大きなファイル一つで管理していたものを小さなファイルに分割して管理すれば、部分的な読み書きで済むようになるし、もし排他制御が避けられない設計であったとしても影響範囲が小さくなる。

しかし、分割したら全体を読み出したい場合には大変にならないかと思うかもしれないが、それほど大がかりなコードにはならない。先程分割したものを復元したければ、

```
$ find stock_qtys -type f | sort | xargs cat↵
```

などと書けばいい。もし、1 列目を消していた場合にそれを復元させたいのであれば、

```
$ find stock_qtys -type f | sort | xargs grep ^ /dev/null | sed 's@^[^:]*/@@' |
sed 's/:/ /'↵
```

と書けばいい。このコマンドの grep〜sed〜sed の部分はファイル名を 1 列目としたい場合に使えるテクニックなので、スニペットとしてそのまま使うといい。一つのデータとして復元したら、在庫数を合計する[*2]なり何なり、あとは自由自在だ。

4.4.2 有効性が低いケースとその対策

さっきから、「有効（な場合がある）」などといまいちスッキリしない書き方をしていたが、それは次のような場合にはあまり有効でないからだ。

- データ全体を頻繁に読み出す必要がある場合

[*2] AWK などで自力で計算スクリプトを書いてもいいが、ShellShoccar コマンドセットの中にある sum-up コマンド群の “sm2” や “sm4”、“sm5” を使うと便利だ。後の版では詳しく紹介したい。

- 行数（レコード数）が 1 万を大きく超える場合

HACK 3.2 での考察からわかるように、ファイルオープンには大きなコストがかかる。「全体を読むのは棚卸しの時くらい」などというように頻繁には読まないならいいが、そうでないなら少し作戦を練るべきだ。分割数を減らすという手もあるだろうし、または頻繁に必要な情報を別途管理するとい手もある。例えば頻繁に読み出す必要のある理由が「全体の在庫数」であるならば、全体の在庫数が格納されたファイルを別途用意し、個々の商品の在庫数を増減させる時に合わせて更新するようにすればいい。

とにかく、データ管理は頭を使って賢くやることだ。

HACK 4.5　加工元データファイルに加工後データを上書かない

上書きは、二度とやり直しできない、履歴が残らないなど他のデメリットも多数ある。

例えば今、次の処理をしなければならないものとする。

0) 外部から ZIP ファイルを受け取る。
1) その ZIP ファイルを展開する。（すると JSON ファイルが出てくる）
2)（JSON 形式そのままだと扱いづらいので）パースして JSONPath-value 形式にする。
3) JSON データ（JSONPath-value でパースしたもの）の中に書き込まれている、発注データをすべて抽出する。

つまり、段階を追ってデータ形式が変化していくようなケースだ。この一連のデータ加工処理をするにあたり、まずは File/Dir Hack の観点から、好ましくない方法と、好ましい方法を図示する。

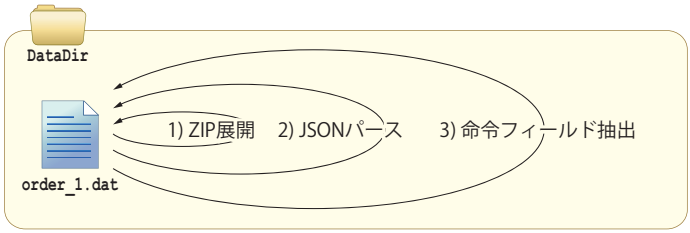


図 4.1 【a】データ加工をする都度、同じファイルに上書きして更新していく

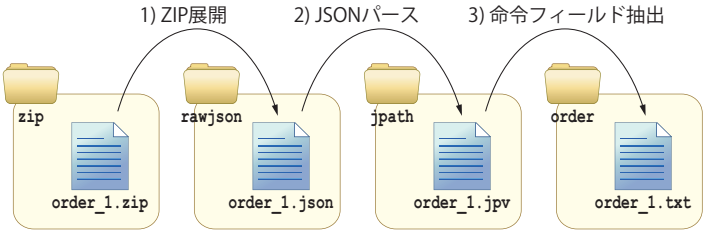


図 4.2 【b】データ加工をする都度、次々と別のファイルに書き出していく

言いたいことは、【a】よりも【b】の方が好ましいということだ。ここでは【b】の方式を「ウォーターフォール書き出し方式」と呼ぶことにする。UNIX では一つのファイルを読み込んでいる最中に同じファイルにデータを書き込もうとするとデータが壊れるので、直接上書きすることは稀だろうが、HACK 4.3 のようにして、加工後のデータを一時ファイルに書き出してから元のファイルに mv することも無闇にやるべきでない。

また、これらの図では、途中の工程で出てくる中間データをすべてファイルに落としているが、その中間状態に保存する意味がないならパイプで繋ぐのでもよい。とにかく、加工が完了したデータを元のファイルに書き戻すか、別のファイルに書き出して終わるかの差である。

4.5.1　ウォーターフォール書き出しが好ましい理由

排他制御が必要になる可能性を減らす

元のファイルに上書きするということは、もし他にも同じ元ファイルを読みたいと思っている相手がいた場合、突然内容が変わってしまう恐れがあるので場合によっては排他制御が必要になってしまう。そういう将来の可能性を作らないという意味でまず、ウォーターフォール書き出しは好ましい。

間違った時にやり直しができる

プログラムのバグや、ユーザーの誤操作、あるいはハードウェアトラブルなどで、加工処理をやり直したいと思う事は往々にして発生するものだ。もし加工後のデータで元ファイルを上書きしてしまったら、二度とやり直しができない。

> ファイルが消えた時、プログラムは仕様書や記憶を頼りに作り直せても、データは二度と作り直せない。

という教訓があるので、肝に銘じておくべきだ。データを消すということは一大決心であるべきといっても大げさではない。

トレースができる

最初のデータが残っていればやり直しができるが、さらに加工処理の推移も要所要所で残してあれば、いつ、どのように加工が行われたか、正しく行われたか、あるいはどこで問題が生じたのかも容易に検証できる。カッコよく言えば「トレーサビリティー」の確保だ。

また、実運用に入る前の開発段階においても、デバッグ効率向上に貢献する。

4.5.2　保管コストとの折り合い

最初のデータや中間工程のデータを残すとそれだけディスク領域を消費する。そのコストとの折り合いは、前述のようなメリットで得られる価値との折り合いで考えるべきだ。File/Dir Hack 的には HACK 1.5 で述べたように、なるべくディスク領域をケチらないように折り合うべきと考える。

HACK 4.2 や HACK 4.3 では在庫数管理のやり方を上書きの処理の例として示した。しかしそれらについても、昨日の分、先週の分、先月の分などのようにして過去の履歴を残しておくべきかもしれない。

HACK 4.6　その他、アトミックな処理を活用する

排他制御を避ける手段は、数あるアトミックな操作を活用すること

ここまで、排他制御を要しない排他処理（トランザクション）として、我々の知っているテクニックを紹介してきたが、あなたのアイデア次第で他にもいろいろなテクニックが生み出せるだろう。ただ、これらすべてのテクニックで共通していることは、アトミックに行われるファイル操作を活用しているということである。

そこで、ファイル操作におけるアトミックな操作を列挙しておく。これらを組み合わせ、あなたにも排他制御不要なトランザクション処理方法を発明してもらいたい。

4.6.1　アトミックと見なせるファイル操作

以下にアトミックと見なして差し支えないファイル操作とコマンドを列挙する。

「見なして差し支えない」という微妙な表現を用いたのには理由がある。例えばディレクトリー削除の rmdir コマンドは、中にある特殊ディレクトリーである “.” と “..” の削除など、内部的には複数の操作から構成されているためにアトミックではないが、途中で邪魔されると不都合が生じるために保護されている。このようなケースが存在するためである。

- 一般ファイルの操作
  - `touch` *regfile*
    - 空ファイルの新規作成・タイムスタンプ更新
  - `echo > "`*str*`"` *regfile*
    - 1 ブロックサイズ（4096 バイト）以下での新規作成
  - `echo >> "`*str*`"` *regfile*
    - 1 ブロックサイズ（4096 バイト）以下での追記
  - `mv` *regfile path*
    - 移動・名前変更（同一パーティション内のみ）
  - `rm` *regfile*
    - 削除
  - `chmod` *regfile*
    - パーミッション変更
  - `chown` *regfile*
    - 所有者変更
- ディレクトリー操作
  - `mkdir` *dir*
    - 新規作成
  - `rmdir` *dir*
    - ディレクトリー削除（空である場合のみ）
  - `mv`、`touch`、`chmod`、`chown`
    - （一般ファイルと同様）
- その他特殊ファイルの操作

- `ln` *regfile link*
  - ハードリンク作成
- `ln -s` *file symlink*
  - シンボリックリンク作成
- `mkfifo` *namedpipe*
  - 名前付きパイプ作成
- `mv`、`rm`、`touch`、`chmod`、`chown`
  - （一般ファイルと同様）

なお、これらのコマンドの操作対象とするファイルやディレクトリーは、1 回のコマンドにつき一つだけであること。ワイルドカードを用いるなどして複数を対象とした場合には、それらのファイル・ディレクトリーに対して一つずる操作が発生し、アトミックにはならない。

第 5 章

排他処理（トランザクション処理）

複数のプログラムやユーザーからのアクセスを受け付けるシステムを設計すると、トランザクション処理、つまりそれをやり始めたら邪魔されることなく最後までやり終えなければならない処理というものが大抵生じる。……という課題に対して、前の章では知恵を絞って排他制御不要にするアイデアをいくつか紹介してきたが、やはりそれでは済まないものもある。

この章では、真正面から排他制御をやる方法を、2 種類（排他ロックと共有ロック）解説する。

HACK 5.1　排他ロックは「ln などの早い者勝ち」ルールで捌く

「最初に誰かがやったら以後は誰も同じことができなくなるファイル操作」に成功した者にアクセス権を与えるというルールを作ればよい。

5.1.1　2 種類のロック

トランザクション処理をさせる際に用いるロックには大きく分けて 2 週類ある。本題に入る前にまず、この二つの違いについて説明しておかなければならないだろう。

- **排他ロック**
  - ある資源（ファイルなど）に対して誰かがロックを掛けたら、他の誰にも同じ資源にロックを掛けさせない仕組み。
  - 資源を自分一人で占有するためのもの。
  - 「（今データを書き換えてるから）他者は書き込み禁止なのはもちろん、読み込みも禁止だ」という場面で用いる。
- **共有ロック**
  - ある資源（ファイルなど）に対して誰かがロックを掛けても、他者は上記の「排他ロック」でさえなければ、同時にロックできる仕組み。
  - 資源を誰かに占有させず、皆で共有できることを保証するためのロック。
  - 「（今データを読み込んでるから）他者は読み込むのは構わないけど、書き込みは禁止だ」という場面で用いる。

C 言語の中でなら、ファイルに対してこれら 2 種類のロックを行ってくれる関数がある[*1]。しかしシェルスクリプトから呼び出せるコマンドにはそのようなものが無い。Linux 独自には flock というコマンドがあることにあるが、ロック（トランザクション区間）の開始端や終了端を宣言するものではなく、flock コマンドというそのコマンドの実行中にだけロックを掛けるという仕様になっていて、これは使いづらい。それにもちろん、Linux 以外の環境で使える保証もない。となれば、ファイルやディレクトリーの仕組みを駆使して何とか同じようなことを考えるしかない。

5.1.2 排他ロックの考え方

本質的な目的とは何か

先程確認したように、排他ロックとは、ファイルなどを他の誰にもいじらせないための手段だ。つまり、ロックする本人にしてみれば「今からこのファイルは他の誰もいじらない」という約束が最低限得られればよいし、他の者にしてみれば「今このファイルはいじっちゃいけないんだ」と分かれば最低限事足りる。そうやってロックに関する合図がやりとりができさえすれば、あとは各々が守ればよいだけだ。

このようにしてロック状態を知らせるだけで、遵守するかどうかはプログラムに依存させる種類のロック機構を**アドバイザリーロック**と言う。じつは、fcntl() 関数や flock コマンドなどの UNIX に存在する主要なロック機構も、このアドバイザリーロックである。

ロックの合図の仕組みを考える

排他ロックは、（一つのファイルなどに対して）誰か一人が掛けられることが重要だ。だから基本的には、最初にロックを希望した者に許可を与え、そのプロセス・プログラムが解除しない限りは、他者は不許可にしなければならない。つまり「早い者勝ち」だ。

そういうルールを実現するのに都合のいい仕組みは無いかと知恵を絞ると……、ファイル・ディレクトリーの仕組みにはこんなにも都合のいいものがあることに気づく。

a. ディレクトリー名を決め（例えば “*lockdir*”）、早い者勝ちでそれを作る。
   - `mkdir` *lockdir*

b. シンボリックリンクのファイル名を決め（例えば “*lockfile*”）、早い者勝ちでそれを作る。（リンク元は何でもいい）
   - `ln -s` *somefile lockfile*

c. ハードリンクのファイル名を決め（例えば “*lockfile*”）、早い者勝ちでそれを作る。（リンク元は何でもいい）
   - `ln` *somefile lockfile*

d. ファイル名を決め（例えば “*lockfile*”）、上書き禁止モード（set コマンドの “`-C`” オプション）にしながら早い者勝ちでそれを新規作成する。
   - `(set -C; : >` *lockfile*`)`

これらはどれも、できあがったディレクトリーやファイルを消さない限り、再実行しても失敗する。もしこれらが成功してしまってはファイルシステムとしての秩序が崩壊してしまうため、OS が監視し、阻止しているのである。

だから、あなたが今アプリケーション上で行おうとしている排他制御にも、この性質を巧みに利用すれば

[*1] fcntl() 関数の第 2 引数に “F_SETLK” などの値（コマンド）を設定した場合。

いい。

5.1.3　排他ロック実装のための具体的なルール

以上の考察を踏まえ、ここでまた新たな決め事を設ける。

【決め事 9】排他制御のためのルール

- 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。

なお、「再度ロックファイル作成を試みる」までの時間は、許容できる待ち時間やシステムの負荷などから臨機応変に決めること。実はこのルールに基づく排他制御というのはよく知られている方法で、CGI スクリプトなどでもよく用いられる。ただ、同時に欠点もよく知られている。それは、プログラムの設計ミスや何らかの異常終了によってロックファイルが消されずに残ってしまうと、それ以降そのロック対象が永遠にロックされたままになってしまうということだ。

残ってしまったものはしょうがないため、次のルールを追加して対処する。

【決め事 10】ロック解除失敗のための追加ルール

- 一定以上古いロックファイルが残っていたら消してよいものとする。
- ただし消す役割は、一つのロックファイルに対して一つのプロセスしか担当してはならない。

「一定以上古い」とは、仕様上あり得ない古さと思える期間のうちで最も短いものである。例えば、どんなに長くとも排他区間の処理に 60 秒も掛からないというなら 60 秒とする。なお、秒単位でのファイル新旧比較は、find コマンドの “`--newer`” オプションと touch コマンド、それとコマンドセット “ShellShoccar” に収録した utconv というコマンドを組み合わせればできる[*2]。

注意しなければならないのは二番目のルールであり、これが必要な理由は次のとおり。

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そうとしている時、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……。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からだ。

一般的には、すべてのロックファイルを一つのディレクトリーの中で管理し、そのディレクトリーの古いプロセスを削除する専門のプロセスを crontab などで立てるとよい。

5.1.4　排他ロックコマンド “pexlock”

この考え方に基づいて排他ロックを自力で実装してももちろん構わないが、コマンドセット “ShellShoccar” には、ここまで述べたアイデアをコマンド化したものを用意してある。

- pexlock – 排他ロックコマンド
  - 排他ロックのために、指定されたロックファイルを（“set -C” 方式で）作成する。失敗した場合、デフォルトでは 1 秒間隔で 10 秒後までリトライする。

[*2] 詳細は次の記事を参照。→「find コマンドで秒単位に新旧比較したい - Qiita」

  - 書式: `pexlock` [*options*] *lockname* [*lockname* ...]
    - *lockname*：ロック名、この名前で排他ロック用のファイルを作る。複数指定可。
  - オプション
    - “`-d` *dir*”：ディレクトリー “*dir*” の中にロックファイルを作る。従って、この場所が違えば同じ名前で別のロックファイルを作ることもできる。無指定の場合は、環境変数 “PLOCKDIR” の値を参照し、それもなければカレントディレクトリーに作る。
    - “`-w` *maxwait*”：ロックに失敗した場合は、最大 *maxwait* 秒間、1 秒毎にリトライする。−1 が指定された場合は成功するまで無限にリトライする。デフォルトは 10 秒。
  - 戻り値
    - 0：指定された一つ以上のロックファイル作成に成功した
    - 0 以外：指定されたすべてのロックファイル作成に失敗した
- punlock – ロック解除コマンド
  - ロック解除のために、指定されたロックファイルを削除する。
  - 書式: `punlock` [*options*] *lockname* [*lockname* ...]
    - *lockname*：ロック名、この名前のロックファイルを削除する。複数指定可。
  - オプション
    - “`-d` *dir*”：ディレクトリー “*dir*” の中で削除対象ロックファイル探す。無指定の場合は、環境変数 “PLOCKDIR” の値を参照し、それもなければカレントディレクトリーを対象とする。
  - 戻り値
    - 0：指定された一つ以上のロックファイル削除に成功した
    - 0 以外：指定されたすべてのロックファイル削除に失敗した
- pcllock – 解除漏れロックファイル削除コマンド
  - ロック解除がなされずに残った明らかに古いロックファイルを削除する。
  - 書式: `pcllock` [*options*]
  - オプション
    - “`-d` *dir*”：ディレクトリー “*dir*” の中で残存ロックファイル探す。無指定の場合は、環境変数 “PLOCKDIR” の値を参照し、それもなければカレントディレクトリーを対象とする。
    - “`-l` *lifetime*”：*lifetime* 秒以上古いファイルを残存ロックファイルと見なす。デフォルトは 600 秒。
  - 戻り値
    - 0：指定された一つ以上の残存ロックファイル削除に成功した
    - 0 以外：指定されたすべての残存ロックファイル削除に失敗した
  - 注意
    - このコマンドは、（同じディレクトリーに対して）複数のプロセス・プログラムから実行してはいけない。crontab ファイルに設定するなどし、一つのプログラムが代表して実行すべきである。

排他ロックの具体例

前の章で例に挙げた在庫数ファイルの更新に応用するとすれば、次のような使い方をする。

■在庫数ファイルのロックに pexlock コマンドを利用する例

```
#!/bin/sh

# --- 異常終了時でも極力ロックファイルを消すためのトラップ関数を定義 ---
exit_trap() {
  set -- ${1:-} $?  # ←この関数の呼び出し時に第1引数が設置された場合、
                    #   最後にその値でexitが実行され、なければ現在の戻り値で
                    #   exitされるようにするためのテクニック
  trap - EXIT HUP INT QUIT PIPE ALRM TERM   # ←トラップ解除
  punlock -d "$Dir_lock" stock_qty 2        # ←終了時に忘れずにロックを解除
  exit $1
}
trap 'exit_trap' EXIT HUP INT QUIT PIPE ALRM TERM

                  :
                  :

# --- 在庫ファイルに排他ロックを掛ける（最大15秒間待つ） ---
pexlock -d "$Dir_lock" -w 15 stock_qty || {
  echo "${0##*/}: Error: Failed to ex-lock 'stock_qty'" 1>&2
  exit 1
}

# ====================================================================
# ↓↓↓ 排他ロック区間ここから

# --- 在庫数ファイル更新シェルスクリプトを実行する ---
latest_stockqty.sh > "$DirData/stock_qty.txt"

# ↑↑↑ 排他ロック区間ここまで
# ====================================================================

# --- 在庫ファイルの排他ロックを解除する ---
punlock -d "$Dir_lock" stock_qty
[ $? -eq 0 ] || echo "${0##*/}: Warning: Failed to unlock 'stock_qty'" 1>&2

                  :
                  :
```

pexlock を実行し、成功したら次の行に進む。そこから punlock コマンドまでが排他ロック区間なので、ここで在庫数の書き換えを行う。終わったら punlock で忘れずにロックを解除する。この例では念のため、

ロック解除失敗時に警告メッセージを出すようにしている。

なお、何らかの理由でロックがされたまま終了することを極力防ぐため、主要シグナルに対して、このシェルスクリプトの終了時に念のため punlock を実行するような関数を冒頭で定義している。

また、このシェルスクリプトとは別に、次のようなコードを crontab ファイルに登録しておく。次のように書いて[*3]登録しておけば、万が一ロック解除を忘れていても 60 秒以内には消えるはずだ。

■残存ロックファイル削除のための crontab ファイルへの追加

```
# 残存ロックファイル削除（60秒以上古いファイルを残存ロックファイルと見なす）
* * * * *  pcllock -d /PATH/TO/Dir_lock -l 60
```

5.1.5 制約事項

もちろんこの方法が完璧だと言うつもりもない。次のような弱点がある。

ロック失敗時のリトライは 1 秒間隔

別のプロセスがロックしている最中で、自分がロックに失敗した場合はしばらく待ってから再度ロックを試みなければならない。POSIX の範囲の sleep コマンドは最小間隔が 1 秒であるため、1 度失敗したら例え長く感じても 1 秒待たなければならない。もちろん、1 秒未満を設定可能な sleep コマンドを使ったり自作する[*4]ならその限りではない。

ちなみに sleep をせずにリトライを繰り返すこともできなくもないが、コンピューターに必要以上の負荷を掛けるので勧められない。

ロックを希望した順番に権利が回ってくるとは限らない

C 言語の範囲、つまりシステムコールで用意されているロック機構には、先行するプロセスががロックを解除した時、ロックを希望して待っている順にロックの権利が回ってくるものもあるが。ここで紹介した方式ではそうはいかない。ロック解除後に、タイミングよく最初にロックを掛けに来たプロセスに権利が渡る。ある電話番号に電話が殺到している時、掛けた順番ではなく、運良く回線の空いているタイミングに電話を掛けた者が繋がるため、運が悪い者はいつまで経っても繋がらないことと同様だ。この現象は**リソーススターベーション（“Resource Starvation”、資源飢餓）**と呼ばれている。

もし順序が大事というのであれば、ロックという方式を諦め、次章で解説する非同期キューイング方式を使うべきだ。

HACK 5.2 共有ロックは、ディレクトリーでメンバー管理する

最大の課題は、複数人で対等にロックを管理する仕組み考えること。

共有ロックの性質と用途については排他ロックのところで既に説明したからよいだろう。というわけで、

[*3] crontab ファイルの中では、シェルスクリプトで使っていたシェル変数の値が共有できないのでロックファイル用ディレクトリーまでの絶対パスを書く必要がある。

[*4] 参照 →「POSIX 原理主義的 1 秒未満 sleep - Qiita」

早速、共有ロックを実現するための考え方の解説に入る。

5.2.1　共有ロックの考え方

共有ロックとは排他ロックから守る仕組みでもある。ゆえに、排他ロックと干渉し、互いに排他的な関係になるようにしなければならない。そしてまた共有ロックは、皆でロックを共有し、しかも最初にロックした者が先に抜けることもあり得るため、ロックという権利の管理が複雑にならざるを得ない。

従って、共有ロックに求められる性質をまとめると次のようになる。

1. 同じ資源（ファイル等）に対し、排他ロックを掛けようとしている者がいたら、それを阻止できること。
2. 一つの資源（ファイル等）に対する共有ロックは、最初に掛けた者と最後に解除する者が別であっても支障なく動作できること。
   - ロック権（ロックファイル）の管理役を誰かに固定してはいけない。（固定してしまうと、管理者は他の誰かが共有ロックに参加している限りロック権を放棄できない）
   - 最初のプロセスがロックファイルを生成し、最後のプロセスがロックファイルを削除するルールを作らなければならない。
   - 自分が最初なのか、最後なのかが簡単に判定できる方法を考えなければならない。

排他ロックを阻止するのは簡単だ。排他ロックファイルが作られるのディレクトリーを共有し、そこに先回りして共有ロック用のファイルなりディレクトリーなり、あるいはリンクを作ればよい。ロックファイル作成者と削除者を分ける方の課題は、ファイル・ディレクトリーの仕組みをどのように利用すれば解決できるのか。

5.2.2　共有ロック管理のためのデータ構造はどうするべきか

共有ロックを、ロックに参加するプロセスで対等に管理するにはまず、次のようなディレクトリー構成をとるとよいだろう。

```
LOCKDIR/                              ← ロックファイルを置くためのトップdir
|
+-- sh_lock1/                         ← 1つの共有ロックのトップdir
|     |
|     +-- sh_lock1/                   ← 同じ名前でサブdirを作る
|           |
|           +-- <日時等の一意な値>.<共有ロック取得中のプロセスaのID>
|           +-- <日時等の一意な値>.<共有ロック取得中のプロセスbのID>
|           :            :                        :
|           :            :                        :
|           +-- .busy                 ← 共有ロックdirの中身処理中を示すロックファイル
|
+-- sh_lock2/                         ← 他の共有ロック管理用dir
+-- sh_lock3/                         ← （内部構造は同じ）
:     :                                        :
:
```

```
:
+-- ex_lock1                    ← 同じ名前空間に共存する
+-- ex_lock2                    ← 排他ロック用のファイル
:                                       :
```

ファイルではなくディレクトリーを用いる

まず、排他ロックの管理にはファイルを用いたが、共有ロックの管理にはディレクトリーを用いる。その中で共有ロックを掛けているプロセス ID 一覧を記憶するためだ。なぜ一覧を記録するために、ファイルではなくディレクトリーを選んだ理由は、HACK 4.4 で説明したとおりだ。すなわち、ディレクトリーはランダムアクセス空間であるから、その直下にプロセス ID を名前としたファイル名を置けば、プロセスがどんな順番で参加や脱退をしても更新が簡単にできる。

なお、たとえディレクトリーであっても、排他ロック用のファイルとはきちんと競合できる。ファイルであろうとディレクトリーであろうと、どちらかが先に存在すれば、後から同じ名前のロックファイル（またはディレクトリー）は作れない。

ディレクトリーを同名で 2 階層にする

ここで示したディレクトリーツリーでは、“sh_lock” という同じ名前のディレクトリーがなぜか親子で存在している。じつはこれは、mv コマンドの副作用を避けるための巧妙な仕掛けを作用させるためのものである。

まず、共有ロックはディレクトリーで管理し、その中にはプロセス ID 名のファイル等を置かなければならないことにしているが、そのようなディレクトリーを、いきなりロックファイル用のトップディレクトリー（LOCKDIR）の中で作るわけにはいかない。なぜなら、作り途中のそれをもし他のプロセスが参照してしまったら、誤動作をするからだ。そこで、一旦、隠しファイル名などで共有ロック用ディレクトリーを作り、完成したら本番の名前に改名する必要がある。

名前の付け替えといえば mv コマンドだが、改名後の名前がディレクトリーとして既に存在していた場合、こちらの希望としては失敗してもらいたいのだが、mv は改名する代わりにそのディレクトリーの中に移動してしまう。

■ディレクトリーを mv すると、名前変更の成否だけ試すつもりが、移動処理になってしまう

```
$ mkdir sh_lock1↵              1) 既存の共有ロック dir を疑似的に作成
$ touch sh_lock1/pid↵
$ mkdir .sh_lock1.tmp↵         2) 新規の共有ロック dir（仮）も疑似的に作成
$ touch .sh_lock1.tmp/pid↵
$ mv .sh_lock1.tmp sh_lock1↵  3) 本番名に改名しようとすると…
$ ls sh_lock1↵                 ← 改名されず既存 dir の中に入ってしまっている!
.sh_lock1.tmp
$
```

一方、親子で 2 階層にしておけば、mv しようとした時、既存のディレクトリーがある場合、処理を失敗させることができる。

■同一ディレクトリー名で親子 2 階層にすれば、mv は成否（移動）だけを試せるようになる

```
$ mkdir -p sh_lock1/sh_lock1↵          1) 作成 (既存)
$ touch    sh_lock1/sh_lock1/pid↵
$ mkdir -p .tmp/sh_lock1/sh_lock1↵     2) 作成（新規・仮）
$ touch    .tmp/sh_lock1/sh_lock1/pid↵
$ mv .tmp/sh_lock1 sh_lock1↵          3) 仮 dir を本番 dir に移動しようとすると失敗する
mv: cannot move '.sh_lock1.tmp/sh_lock1' to 'sh_lock1/sh_lock1': Directory not
empty
$
```

共有ロック用ディレクトリーの中を操作する際は、そこに排他ロックを掛ける

共有ロックを掛けるメンバーが増減する場合は、共有ロック用のディレクトリー内にあるファイルの追加・削除、あるいは共有ロックディレクトリー自体の作成・削除が必要になる。しかし、それらのファイル操作にはアトミックと見なせないものが含まれているため、どうしても排他制御が必要になる。そこで、共有ロック用のディレクトリー内に、新たな排他ロックファイルを作る。それが、ディレクトリーツリーの中に記した ".busy" というファイルである。

共有ロックを追加や削除をする時はまず、HACK 5.1 の要領でこのファイル作らなければならないものとし、作業が終わったら速やかに削除するようにする。

5.2.3　共有ロック管理のための各種ファイル操作

次に、そのような構造を持ったディレクトリの操作方法を解説していく。以降で、共有ロック操作のためのシェルスクリプトの例を示す（解説のため一部簡素化している）。

共有ロック用ディレクトリー新規作成

共有ロック用ディレクトリーそのものが無い場合には新規作成する必要がある。既に述べたが、作成途中のものが見られないよう、隠しファイルとしてこっそり生成し、完成しなければならない。

■新規作成のコード例

```
# === 各種定義 ===
Dir_lock=(ロックファイルを置くディレクトリーのパス)
name=(共有ロック名)
ppid=$(ps -Ao pid,ppid                    |  1) ロックを掛けたプロセスの ID を取得
       awk '$1=='"$$"'{print $2;exit}')
now=$(date '+%Y%m%d%H%M%S')                  2) 現在日時を取得
uniqid=$now.$$.$ppid                         3) 自分の ID を加えたユニーク値を作成
Dir_tmp="$Dir_lock/.tmp.$$/$name/$name"      4) 仮作成ロック dir のパスを作成

# === 新規作成処理 ===
```

```
mkdir -p "$Dir_tmp"          || exit 1            5) 仮作成ロック dir を作成
touch    "$Dir_tmp/$uniqid" || {                 6) 共有ロックメンバー個別ファイルの作成
  rm -rf "$Dir_lock/.tmp.$$"
  exit 1
}
mv "$Dir_lock/.tmp/$name" "$Dir_lock/$name" && { 7) 本番 dir として移動
  echo "$Dir_lock/$name/$name/$uniqid"            8) 成功時は個別ファイルのパスを返し
  rmdir "$Dir_tmp"                               9) 仮作成用 dir を削除して
  exit 0                                         10) 終了
}
（mv に失敗したら既存の共有ロックがあるかもしれないので、次のメンバー追加処理を試みる）
```

なお、ディレクトリーの中身を操作しているが作業の排他ロックを宣言するための “.busy” を作る必要はない。なぜなら、仮作成ロック dir であるうちは自分しか触れないし、本番 dir に移動しても移動した時点で作業が完了しているからである。

共有ロック用メンバー追加

既に同じ名前の共有ロックが存在する場合（先程の新規作成に失敗した場合）は、その共有ロック用ディレクトリーに排他ロックを掛けて（“.busy” ファイルを置いて）、共有ロックメンバー個別ファイルを作成する。終わったらロックを解除して、その個別ファイルのパスを返す。

排他ロックに失敗した場合は、そこでリトライをするより戻る最初（新規作成）からリトライする方がよい。なぜなら、共有ロックが busy である可能性はごく稀で（生存時間が短いので）、実は共有ロックではなく排他ロックであったという可能性の方が高いからだ。最初に戻ったら 1 秒 sleep するなどしてから、リトライする。

■共有ロック用メンバー追加のコード例

```
（新規作成ルーチンからの続き）

# === メンバー追加処理 ===
(set -C; : > "$Dir_lock/$name/$name/.busy") ||   1) 排他ロックを試み、
continue                                            失敗したら最初へ戻り、リトライ
touch "$Dir_lock/$name/$name/$uniqid" || exit 1  2) 共有ロックメンバー個別ファイルの作成
rm "$Dir_lock/$name/$name/.busy"                 3) 共有ロック解除
echo "$Dir_lock/$name/$name/$uniqid"             4) 個別ファイルのパスを返して
exit 0                                           5) 終了
```

共有ロック用メンバー削除

一方、共有ロックから抜ける場合の処理では、まず抜けたいと思っている自分が与えられているメンバー個別ファイルを消す。その際もちろん、事前に排他ロックを掛けて（“.busy” ファイルを置いて）おかなければならない。

■共有ロック用メンバー削除のコード例

```
# === 各種定義 ===
memberpath=(新規作成・追加の時に返された共有ロックメンバーファイルパス)
s=${memberpath%/*}; name=${s##*/}                1) パスに含まれる共有ロック名を抽出
s=${s%/*}; Dir_lock=${s%/*}                      2) 同様にして、ロックファイル dir 名も抽出

# === メンバー削除処理 ===
(set -C; : > "$Dir_lock/$name/$name/.busy") || 3) 排他ロックを試み、
continue                                            失敗したら最初へ戻り、リトライ
rm "$memberpath" || exit 1                       4) 共有ロックメンバー個別ファイルを削除

# === メンバー数確認 ===
n=(ls -1f "$Dir_lock/$name/$name" | wc -l)       5) メンバー数を調べる
[ $n -gt 0 ] && {                                   6) 自分以外にメンバーがいたら
  rm "$Dir_lock/$name/$name/.busy"                  7) ロックを解除して
  exit 0                                            8) 終了
}
（メンバーがいなかったら、共有ロック用ディレクトリー削除ルーチンへ）
```

共有ロック用ディレクトリー削除

共有ロックメンバー数が 0 のディレクトリーはもはや存在する意味がないばかりか、残っていては同名の排他ロックが掛けられなくなるので削除しなければならない。

これは先程のメンバー削除の続きであれば簡単で、対象ディレクトリーを隠しファイル名などに改名したうえで削除すればよい。なお、ロックファイルも一緒に消えるので特にロック解除処理をする必要はない。

■共有ロック用ディレクトリー削除のコード例

```
# === ディレクトリー削除処理 ===
mv "$Dir_lock/$name" "$Dir_lock/.tmp.$$" &&  1) ディレクトリーを改名
rm -rf "$Dir_lock/.tmp.$$"               &&  2) 改名したディレクトリーを削除
exit 0                                       3) 終了
```

ところで、改名せずにいきなり共有ディレクトリーを消してもほとんどの場合で問題は起こらないだろうが、念のためである。もし、内部のロックファイルが消されたのに、まだ共有ロックディレクトリーが残っているという状態が他のプロセスに見えてしまったら、不具合が起こってしまう。

5.2.4　共有ロックコマンド “pshlock”

排他ロックと同様に、共有ロックについてもここまで述べた考え方に基づいたコマンドを作った。コマンドセット “ShellShoccar” に用意してある。

- pshlock – 共有ロックコマンド
  - 共有ロックのために、指定された名前でロック用ディレクトリーを作成する。通常、同名の排他ロックが存在しなければ成功し、ロックメンバー個別ファイル（ロック解除時に必要）のパスが返される。失敗した場合、デフォルトでは 1 秒間隔で 10 秒後までリトライする。

  - 書式: `pshlock` [*options*] *lockname* [*lockname* ...]
    - *lockname*：ロック名、この名前で共有ロック用のファイルを作る。複数指定可。
  - オプション
    - “`-d` *dir*”：ディレクトリー “*dir*” の中にロックファイルを作る。従って、この場所が違えば同じ名前で別のロックファイルを作ることもできる。無指定の場合は、環境変数 “PLOCKDIR” の値を参照し、それもなければカレントディレクトリーに作る。
    - “`-w` *maxwait*”：ロックに失敗した場合は、最大 *maxwait* 秒間、1 秒毎にリトライする。−1 が指定された場合は成功するまで無限にリトライする。デフォルトは 10 秒。
    - “`-n` *maxsharing*”：共有ロックに参加できる数を、最大 *maxsharing* までとする。このオプションは、次の HACK 5.3 のためのもの。負値は無制限を意味する。デフォルト値は −1 であり、つまり無制限。
  - 戻り値
    - 0：指定された一つ以上のロック用ディレクトリー作成に成功した
    - 0 以外：指定されたすべてのロック用ディレクトリー作成に失敗した
    - stdout：ロックメンバー個別ファイルのパス（ロック解除時に必要）
- punlock – ロック解除コマンド
  - ロック解除のために、指定されたロックファイルを削除する。
  - 書式: `punlock` [*options*] *memberfile* [*memberfile* ...]
    - *memberfile*：ロックメンバー個別ファイルのパス（ロック解除時に必要）。共有ロックの場合はロック名ではなく、pshlock コマンドから与えられたファイルパスを渡さなければならない。複数指定可（解除したい排他ロックと一緒の指定も可）。
  - オプション
    - “`-d` *dir*”：【排他ロック解除も併用する時のみ必要】ディレクトリー “*dir*” の中で削除対象ロックファイル探す。無指定の場合は、環境変数 “PLOCKDIR” の値を参照し、それもなければカレントディレクトリーを対象とする。
  - 戻り値
    - 0：指定された一つ以上のロック用ディレクトリー削除に成功した
    - 0 以外：指定されたすべてのロック用ディレクトリー削除に失敗した
- pcllock – 解除漏れロックファイル削除コマンド
  - ロック解除がなされずに残った明らかに古いロックファイルを削除する。
  - （使い方は排他ロックの時とまったく同じ）

共有ロックの具体例

前の章で例に挙げた在庫数ファイルの更新に応用するとすれば、次のような使い方をする。

■在庫数ファイルの共有ロックに pshlock コマンドを利用する例

```
#!/bin/sh

# --- 異常終了時でも極力ロックファイルを消すためのトラップ関数を定義 ---
exit_trap() {
  set -- ${1:-} $?  # ←この関数の呼び出し時に第1引数が設置された場合、
                    #   最後にその値でexitが実行され、なければ現在の戻り値で
```

```
                    #   exitされるようにするためのテクニック
  trap - EXIT HUP INT QUIT PIPE ALRM TERM   # ←トラップ解除
  punlock -d "$Dir_lock" stock_qty 2        # ←終了時に忘れずにロックを解除
  exit $1
}
trap 'exit_trap' EXIT HUP INT QUIT PIPE ALRM TERM

                    :
                    :

# --- 在庫ファイルに排他ロックを掛ける（最大15秒間待つ） ---
mylock=$(pshlock -d "$Dir_lock" -w 15 stock_qty)
case $mylock in '')
  echo "${0##*/}: Error: Failed to sh-lock 'stock_qty'" 1>&2
  exit 1
;; esac

# ====================================================================
# ↓↓↓ 共有ロック区間ここから

# --- 在庫数ファイルを参照する ---
num_of_products=$(wc -l "$DirData/stock_qty.txt" | tr -cd '[0-9]')

# ↑↑↑ 共有ロック区間ここまで
# ====================================================================

# --- 在庫ファイルの排他ロックを解除する ---
punlock "$mylock"
[ $? -eq 0 ] || echo "${0##*/}: Warning: Failed to unlock 'stock_qty'" 1>&2

                    :
                    :
```

pshlock を実行する際は、標準出力に返ってくる変数の結果をシェル変数に受け取るようにする。（複数の共有ロックを掛ける場合は複数行返されるので適宜処理すること）ロックが一つだけの場合は、変数が空文字かどうかでロックの成否判定をすればよい。

ロックに成功したら次の行に進む。そこから punlock コマンドまでが排他ロック区間なので、ここで在庫数の参照を行う。終わったら punlock で忘れずにロックを解除する。この例では念のため、ロック解除失敗時に警告メッセージを出すようにしている。

なお、冒頭にあるトラップ関数はロック解除漏れ防止のためであり、HACK 5.1 の時と同じだ。

また、その時と同様、次のようなコードを crontab ファイルに登録しておく（排他ロックの時に登録してあれば不要）。次のように登録しておけば、万が一ロック解除を忘れていても 60 秒以内には消えるはず

である。

■残存ロック削除のための crontab ファイルへの追加

```
# 残存ロック削除（60秒以上古いファイルを残存ロックと見なす）
* * * * *  pcllock -d /PATH/TO/Dir_lock -l 60
```

HACK 5.3 セマフォ制御は、共有ロックに最大共有数を設けて対応する

共有ロックをセマフォ制御に流用できるのも、**File/Dir Hack** で自分たちで自作してきた強み

5.3.1 セマフォとは

セマフォについておさらいしておこう。これは、スーパーやコンビニのレジ待ちを思い浮かべると分かりやすいかもしれない。

忙しい時には大抵複数のレジで会計ができるが、客がレジの台数以上いて、どのレジも空いていない状態の時、「セマフォの値は 0」である。そして、ようやく客が減ってきてレジが一つ空いた時、「セマフォの値は 1」である。つまりセマフォとは、求めている資源（この場合はレジ）が今いくつ利用可能かを示すための変数である。

コンピューターでも、対応できる装置の数に限りがある場合、この概念が必要になる。例えば、4 コアの CPU を搭載したコンピューターで、たくさんある計算リクエストをとりあえず 4 つだけ同時に受け付けて、一つ終わったらまた新たな一つのリクエストを受け付ける……、といった場合にはセマフォ制御が必要になる。

5.3.2 ファイル・ディレクトリーを駆使して実現する

想像できると思うが、共有ロックで考えたデータ構造、およびファイル・ディレクトリーの取り扱い方を応用すれば簡単にできる。一つの共有ロックに参加できる数を制限すればよい。

具体的には HACK 5.2 で「共有ロック用メンバー追加のコード例」を次のように改良する。

■共有ロック用メンバー追加のコード例（セマフォ制御対応版）

```
maxsharing=(最大共有ロック数を指定しておく)
（新規作成ルーチンからの続き）

# === メンバー追加処理 ===
(set -C; : > "$Dir_lock/$name/$name/.busy") ||   1) 排他ロックを試み、
continue                                            失敗したら最初へ戻り、リトライ
n=(ls -1f "$Dir_lock/$name/$name" | wc -l)       2) 現在の共有数を調べ
[ $n -ge $maxsharing ] && {                      3) 最大共有数に達していた場合は
  rm "$Dir_lock/$name/$name/.busy"                  4) ロックを解除して
```

```
  continue                                              5) 最初へ戻り、リトライ
}
touch "$Dir_lock/$name/$name/$uniqid" || exit 1  6) 共有ロックメンバー個別ファイルの作成
rm "$Dir_lock/$name/$name/.busy"                      7) 共有ロック解除
echo "$Dir_lock/$name/$name/$uniqid"                  8) 個別ファイルのパスを返して
exit 0                                                9) 終了
```

追加したのは 2)〜5) の部分であり、最大共有ロック数に達していたら新たな共有ロックをせずに、リトライする。

セマフォ制御の具体例

既に記したとおり、pshlock コマンドは最大共有ロック数に対応している。書式は 5.2.4（セマフォ用途には “-n” オプションが重要）に記してあるので参考にしてもらいたい。

ここでは具体例を見ていく。といっても共有ロックの時の例とほぼ同じで、pshlock に “-n” オプションを追記した程度だ。

■在庫数ファイルの共有ロックに pshlock コマンドを利用する例

```
#!/bin/sh

# --- 異常終了時でも極力ロックファイルを消すためのトラップ関数を定義 ---
exit_trap() {
  set -- ${1:-} $?  # ←この関数の呼び出し時に第1引数が設置された場合、
                    #   最後にその値でexitが実行され、なければ現在の戻り値で
                    #   exitされるようにするためのテクニック
  trap - EXIT HUP INT QUIT PIPE ALRM TERM   # ←トラップ解除
  punlock -d "$Dir_lock" stock_qty 2        # ←終了時に忘れずにロックを解除
  exit $1
}
trap 'exit_trap' EXIT HUP INT QUIT PIPE ALRM TERM

                  :
                  :

# --- 最大数4で共有ロックを掛け、最大4のセマフォ制御をさせる ---
mylock=$(pshlock -d "$Dir_lock" -w 15 -n 4 cpu_share)
case $mylock in '')
  echo "${0##*/}: Error: Failed to sh-lock 'cpu_share'" 1>&2
  exit 1
;; esac

# ====================================================================
# ↓↓↓ セマフォ区間ここから
```

```
（ここで何かCPU1コアを占有するような処理をする）

# ↑↑↑ セマフォ区間ここまで
# ========================================================================

# --- セマフォを解除する ---
punlock "$mylock"
[ $? -eq 0 ] || echo "${0##*/}: Warning: Failed to unlock 'cpu_share'" 1>&2

                :
                :
```

もちろん、ロックの解除ができなかった時のために、crontab で残存ロックを削除するための設定も忘れずに。

■残存ロック削除のための crontab ファイルへの追加

```
# 残存ロック削除（60秒以上古いファイルを残存ロックと見なす）
* * * * *  pcllock -d /PATH/TO/Dir_lock -l 60
```

5.3.3 リソーススターベーション問題

排他ロックのところでも説明したが、この方式にはリソーススタベーションが起こり得るという問題がある。つまり、リクエストした順番にロックやセマフォ制御の権利が得られるとは限らず、いつまでも待たされる可能性があるということだ。

ただし、リクエストを受け付ける側からすれば可能な限りの能力でリクエストを消化しているのだから、いつまでも待たされるプロセスが出てしまうということは、ほとんどの場合、能力不足である。だから、リクエストを減らすよう工夫・努力するか、能力を上げる方が先だ。

もし、順番通りにリクエストが受け入れられることが重要だという場合は、本章で見てきたロック方式は諦め、次章で紹介する非同期キューイング方式で処理すべきである。

第 6 章

非同期キューイング – 食券方式な食堂の注文管理

前の章でも指摘していたように、ロックによる排他制御ではリソーススターベーション問題、つまり、ある電話番号に電話が殺到している時にいくら早く電話しても、タイミングが良くなければ永遠に繋がらない、という時と同様の問題が生じてしまう。

このような時の運の要素を排し、早くリクエストした者から順番に捌くようにするためにはキューイング機構を考えなければならない。本章では File/Dir Hack 的にそれをどう実現するかを解説する。

HACK 6.1　キューイングは、食券方式の食堂を真似する

食券に携わる人や設備を観察し、ファイル・ディレクトリーで表現すればいい。

6.1.1　飲食店の食券はどういう仕組みか

飲食店で食券と呼ばれる券を買って料理に引き換えるという仕組みは、世界的には珍しいという[*1]。しかしこの仕組みは、要求を順番に、かつ効率良く捌く必要があるシステムにとっては、見習うべき特徴を持っている。そこで、食券方式を採用する飲食店で、注文がどのように管理されているかを観察してみる。

街の小さな飲食店など、ほとんど一人で切り盛りしているような食堂ではだいたい図 6.1 のようになっているだろう。

この図を見ながら、まず登場人物を確認する。コンピューターで言えばプロセスに相当する概念だ。

1. 客
   - 食券を買って、カウンターに置く人。
   - 複数いる。
   - 順番を守ってほしいと思っている。
   - 料理を受け取って食べたら退店。（繰り返さない）

[*1] 食券を導入して代金前払いにしてしまうと給仕係がチップを貰えないため。

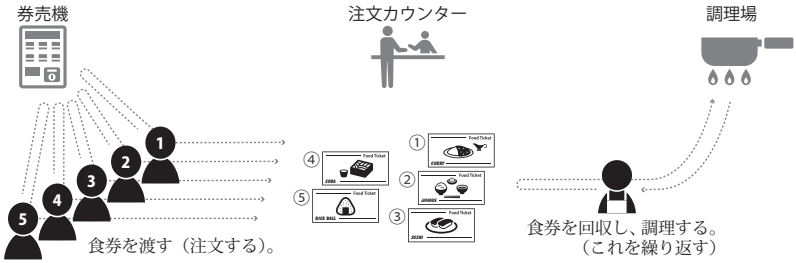


図 6.1 食券方式の食堂（小規模な場合）

2. 店の主人
   - 食券を順番に回収し、調理・配膳まで、店側の作業をすべて担う人。
   - 配膳まで終えたらまた食券の回収からの作業を繰り返す。
   - 基本的には一人。

今さら詳しく説明する必要もないだろう。ただ、店側の担当者は食券の回収と実作業（調理など）のすべてを担当する点に留意しておいてもらいたい。

次に、設備を確認する。これはコンピューターで言えば、プログラムやデバイス、あるいはファイル・ディレクトリーなど、プロセスが利用する対象物である。

1. 食券機
   - 食券を発行する機械。
   - 台数は任意。
2. 受け渡しカウンター
   - 食券を、客から店員へ渡すための置き場。
   - 一箇所。
   - 客がここに食券を置くと、店員は自分のタイミングで回収する。
   - 通常、ここに置かれた順番に回収・調理されるルールになっている。
3. 調理場
   - 食券に記された注文内容に応じて、店員が調理作業を行う場所。
   - もし必要なら複数あってもよい。

今の議論で最も重要なことは、受け渡しカウンターの仕様である。一箇所であること。もし複数個所あったら順番通りに正しく処理するのに手間がかかってしまう。また、券を使うことから、客が注文を与えるタイミングと店主がそれを受け取るタイミングを個別化できる。これが食券方式の大きな利点の一つで、このような形態を非同期式と言う。

同期式と非同期式

食券を観察すると、「客が料理を手に入れる」という一つの目的を達成を達成するのに、非同期式の利点を最大限に活かした賢いやり方であることが見えてくる。

前の章では排他制御（ロック）を散々説明してきたが、あれは同期式に分類されるものであり、両者を比較すると次の表のようになる。

方式	立場	やること
同期式 （ロック）	客	1)　調理場の占有を試みる。使用中なら空くまでじっと待つ（他のことをしていると空いたことに気づけないため）。 2)　自分で調理・配膳する。 3)　調理場を明け渡す。
非同期式 （食券）	客	1)　食券を買ってカウンターに置く。 2)　待つ。待っている間、スマホを弄るなり、仲間と仕事の話をするなり、他のことをしていてもよい。 3)　料理を受け取る。
	店主	1)　食券を回収する。 2)　調理する。 3)　できた料理を配膳する。

同期式の場合は店主が登場せず、完全なセルフサービスだ。敢えて身近な例を挙げるなら、コンビニにある自由に使ってよい電子レンジまたはコーピーマシンかもしれない。客は装置を占有し、自らが調理をするが、別の客が使っている間はじっと待たなければならない。スマホをいじるなどしてボヤボヤしていると、空いた時に後ろで待っている客に文句を言われたり、先に使われるかもしれないからだ。待っている間は完全に時間の無駄になってしまう。

一方の非同期式では、客と店主というように、注文者と調理者の役割分担ができている。代わりに要求を仲介する手段が食券である。こちらの場合は、客は食券を出した後、テーブルに着いて待つ間、別のことをしていられる。

後者の場合、店主が食券が置かれていることに気づいて回収するまでの時間で無駄が発生すると言えなくないが、**混雑度合が激しければ激しいほど非同期であるこちらの方式の方が有利**であることは容易に想像できるだろう。ゆえに、システム開発で排他制御を必要とする場面に遭遇したら、ロック方式だけではなく非同期キューイング方式も検討すべきだ。

6.1.2　食券方式をファイル・ディレクトリーで再現する

あとは、ここまで観察した食券方式をモデル化し、都合よくファイル・ディレクトリー、あるいはプログラムに写像すればいい。

立場	食堂の世界		コンピューターの世界
頼むもの	大勢の客	→	各クライアントプロセス
	食券機	→	クライアントプログラム
仲介するもの	一つ一つの食券	→	一つ一つのチケットファイル
	カウンター	→	一つのチケット授受ディレクトリー
受けるもの	一人の店主	→	一つサービスプロセス（常駐して要求受付）
	調理場	→	サービスプログラム（要求受付・処理）

準備（チケット授受ディレクトリーの作成）

食券方式に習うなら、チケットファイルを授受する場所を用意しておかなければならない。といっても、単に一つのディレクトリーを作成しておくだけである。これはインストーラーのプログラムに作成さるか、システムを稼働させる前に手作業で作る。

■チケットディレクトリーを作成する

```
$ mkdir /PATH/TO/TICKET↵ （例としてこのような名前にした）
$
```

クライアントプログラム

要求を出すプロセスに実行させるプログラムでは、最初にチケット授受ディレクトリーの場所を定義しておき、あとは必要な場所で、そのディレクトリーにチケットファイルを書き出すだけである。

ただし、具体的には 2 通りのやり方がある。まず、要求内容がファイル名のみで事足りるようなシステムの場合だが、この場合は次のようにしてチケット授受ディレクトリーの中に空のチケットファイルを直接作成すればよい。空ファイルの作成はアトミックに実行されるからだ。

■チケット発行ルーチンの例（ファイル名だけでよい場合）

```
Dir_ticket=/PATH/TO/TICKET

            :
            :

order=ここに依頼内容をファイル名として書く"
uniqid="$(date '+%Y%m%d%H%M%S').$$"        ←ファイル名衝突回避用の ID
: >"$Dir_ticket/$order.$uniqid" || exit 1

(必要に応じて続きの処理)
```

ここで注意しなければならないのは、ファイル名にはユニークな ID を含めること。そうしないと、誤って他のチケットファイルを消してしまう恐れがあるからだ。ユニークな ID の元としては、現座日時と自プロセスを含めておけばよいだろう。

さて、もう一つのチケットファイル生成方法、要求内容がファイル名だけでは伝えきれない場合に必要になる。この時は、チケット授受ディレクトリー内に一旦隠しファイルとしてファイルを作成し、終わったら mv コマンドで隠しファイルでない名前に改名する。

■チケット発行ルーチンの例（ファイル内に詳細な指示を書きたい場合）

```
Dir_ticket=/PATH/TO/TICKET

            :
            :
```

```
uniqid="$(date '+%Y%m%d%H%M%S').$$"                ←ファイル名衝突回避用の ID
: > "$Dir_ticket/.$uniqid" || exit 1               ←隠しファイルとして新規作成
echo "（指示内容 1） >> "$Dir_ticket/.$uniqid"
echo "（指示内容 2） >> "$Dir_ticket/.$uniqid"
          :
echo "（指示内容 n） >> "$Dir_ticket/.$uniqid"
mv "$Dir_ticket/.$uniqid" "$Dir_ticket/$uniqid"  ←改名し、正式なチケットにする

（必要に応じて続きの処理）
```

ファイル内に指示内容を書く時に最初の方法でやってしまうと、チケットを作りかけの状態でサービスプログラムにチケットを回収される恐れがある。この教訓は HACK 4.3 で既に説明済だ。

あなたはもしかすると、

> ファイルに書きたい指示内容が、1 ブロックサイズ（4096 バイト）以下だったらアトミックであるから、
>
> ```
> echo "（指示内容）" > "$Dir_ticket/$uniqid"
> ```
>
> のように書いてもよいのではないか。

と思うかもしれない。しかし、厳密にはファイルのオープン・書き込み・クローズという 3 つの工程があってアトミックではないから、運が悪ければ作りかけの状態で処理されてしまうかもしれない。

サービスプログラム

一方、今度は店主側、つまりチケットファイルを回収して処理する方のプログラムの例だ。

■チケット処理ルーチンの例（処理者が自分だけの場合）

```
Dir_ticket=/PATH/TO/TICKET

          :
          :

while :; do                                        ←チケット処理用の無限ループ開始端
  ticket=$(ls -cr1 "$Dir_ticket" | head -n 1)      1) 最古のチケットファイルを選ぶ
  case "$ticket" in '') sleep 1;continue;; esac    2) チケットが無ければ最初へ戻る
            :                                      3) 適宜チケットを処理する
  （チケットファイルに応じた処理）
            :
  rm "$Dir_ticket/$ticket"                         4) 処理済チケットを削除する
done
```

店主が、食券の回収・調理を繰り返していたように、サービスプログラムは、チケットの回収・処理を繰り返すために無限ループを作る。シェルスクリプトの場合、具体手には while ループを使うとよいだろう。

ループの中ではまず、今存在するチケットの中で一番最初に与えられたものを一つ選ばなければならない。食券カウンターのルールに忠実にすることを考えるなら、この時、チケットファイルが作り終えられた日時ではなく、チケットファイルが授受用ディレクトリーに置かれた日時を見るべきであり、その場合は ls コマンドの “-c” オプション（ctime[*2]の新しい順に並べる）が役に立つ。併用しているその他のオプション “-r1” は、「並べ替えの順を反対（つまりこの場合古い順）にし、1 ファイル 1 行で表示せよ」という意味である。ただ、チケットファイルが一つも存在しない場合には何も取れないので次の行でチェックし、無かったら過剰なループを避けるためにしばらく待った後に最初へ戻る。

ファイルが一つ取り出せたらチケットを処理し、最後にそのファイルを削除して一回のループが完了する。

さて、サービスプロセスの負荷を減らすべく、それを複数にすることも可能だ。ただし、その場合はサービスプログラムにもう一工夫必要になる。

■チケット処理ルーチンの例（処理を複数プロセスで行う場合）

```
Dir_ticket=/PATH/TO/TICKET
Dir_tmp=/tmp                    ←チケット回収用のディレクトリーを新たに用意する

          :
          :

while :; do                                          ←チケット処理用の無限ループ開始端
  ticket=$(ls -cr1 "$Dir_ticket" | head -n 1)          1) 最古のチケットファイルを選ぶ
  mv "$Dir_ticket/$ticket" "$Dir_tmp" || {             2) チケットファイルの回収を済ませる
    sleep 1                                               もし失敗したら、しばらく待って
    continue                                              最初へ戻る
  }                                                                :
                  3) 適宜チケットを処理する
   （チケットファイルに応じた処理）
               :
  rm "$Dir_tmp/$ticket"                              4) 回収済チケットを削除する
done
```

先程のプログラムとの違いは、クライアントにチケットを置いてもらう授受ディレクトリーとは別に、回収するためのディレクトリーも用意するという点である。回収用ディレクトリーは自分で作ってもよいし、“/tmp” などの既存ものを使ってもよい。なぜ、最初に回収が必要なのか。理由は、複数いる他のサービスプロセスも一緒に同じチケットを処理しまっては困るからだ。だから、回収に成功したことを確認した後に処理作業に入る。もし回収に失敗したらループの最初に戻るようにする。

このようにしてサービス側のリソースを増強して、負荷分散を図ることができる。

[*2] ctime はファイル属性が変更された日時を示すが、mv を実行した場合にも変更される。

HACK 6.2　規模や複雑さが増したら、大規模な食堂の仕組みを真似る

学食などの大規模な食堂を観察すれば、店側の人数も多く、役割も増えているのがわかる。

先の HACK では、比較的規模の小さい非同期キューイングの事例を見てきた。しかし例えば、処理としてやらなければならない工程がたくさんあったり、多岐に渡るなどした場合、一つのプロセスが食券の回収からそれら複雑な処理のすべてを担うのは避けたいと考えるかもしれない。そのような場合は、少し工夫してみるべきだ。

6.2.1　大規模な食堂はどうなっているか

学生や大企業の社員をやってないとお目にかかる機会が少ないが[*3]、学生食堂や社員食堂など何百人以上も来店するような食堂の場合、店側の設備も少しアレンジされている。

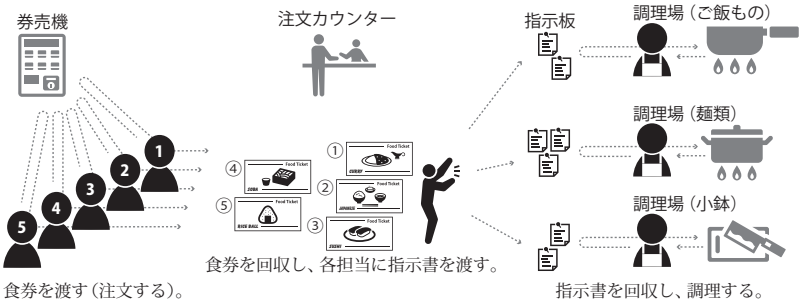


図 6.2 食券方式の食堂（大規模な場合）

図 6.2 はその一例である。小規模な時の違いは次のとおりだ。

1. 調理をせず、食券の回収と指示だけ行う係がいる。
2. 調理する係は、麺類、ご飯もの、小鉢などのように、料理の種類によって分担されている。
3. 食券を受け取る係から調理する係への指示には、また別の札を使っている。

実際の大規模食堂には様々なパターンがあるが、とりあえずこのようにモデル化した。実際には、食券カウンターが麺類用とご飯もので分かれている食堂があったり、小鉢は食券無しで後払いだったりといった変則的な食堂もあるが、そういうものは考えないことにする[*4]。

[*3] 有明の近くにテレコムセンターという駅があって、近所に「青海食堂」という食券方式の大きめの食堂があるぞ。もちろん誰でも入れる。

[*4] 特に、麺類とご飯ものの食券カウンターが別という食堂はあまり美しくない設計に思う。そういう食堂を初めて訪れた時は混乱した。

定食を頼めばどの定食にも必ず付く品物（味噌汁やお新香）があったり、うどんやそばばトッピングが違ってもそれぞれ麺は同じだったりするため、そういった食材は、注文ごとに作るよりもまとめて作る方が効率が良いので、大規模店ではなおさら役割を細分化したくなる。システム開発においても、規模や複雑さが増したら同じように考えたくなるはずだ。役割を細分化するとなったら、まず目を付けるのは食券を回収する係と調理する係の分担だろう。回収係は例えば、そばの食券が多いと判断したら麺類調理係に対し、そばを多めに茹でるように指示する。

6.2.2　ファイル・ディレクトリーで再現する

モデル化したら、これを再びファイル・ディレクトリー、そしてプログラムに写像すればいい。

小規模な食券方式の写像ができたのならあとは簡単で、単にチケットファイルを授受する箇所を二段構成にすれば完成だ。簡単なアレンジに過ぎないのでここではプログラム例は示さず、考え方のみを示すが、仮に図 6.2 の構成を素直に写像するのであれば、チケット授受ディレクトリーを四つに増やし、サービス側のプロセスも、クライアントからチケット受け取るものと、実作業用の三つのプロセスという計四つのプロセスにすればいい。また、チケット回収係と実作業係の間でやりとりされるチケットは、クライアントから受け取るものとはまた別に発行する。

段数はいくらでも増やせるが、遅延に注意

たくさんの UNIX コマンドをパイプで繋いで複雑な処理を実現できるのと同様に、チケット授受の段数を増やせば複雑なキューイングにも対応可能だ。しかも UNIX のパイプと違って、流れてくるタスクの分岐が容易だ。

もちろん、物には限度というものがある。いたずらに段数を増やせばそれだけ、チケット授受（つまりタスク伝達）の遅延が増加する。これは、モデルにした人の作業でも同じことだ。非同期処理の欠点でもある。その点を理解しながら設計しなければならない。

第 7 章

SQL to UNIX マイグレーション (1) – コマンド

データ管理といえば、Oracle や MySQL その他のリレーショナルデータベース製品を導入し、それを SQL という言語によって操作するというスタイルに慣れている人が多いだろう。この章からは、そのようにして SQL 流のやり方に馴染んでいる読者向けに、SQL の各「句」は UNIX の世界に翻訳（=UNIX コマンドで実装）するとこうなるという例を様々紹介していく。

まず最初は、SELECT、UPDATE、INSERT といった、SQL 文（";" で終わるまでの一文全体のことで「ステートメント」と呼ばれる単位）の冒頭に来る各コマンドについての置き換えを例示する。

なお、POSIX 標準の UNIX コマンドのみならず、HACK 1.7 で紹介した "ShellShoccar" コマンドセット（Tukubai コマンド含む）も登場するので、その HACK を参照して準備しておくこと。

HACK 7.1　SELECT コマンド

SELECT コマンドの冒頭にある SELECT 句は、列（列またはフィールド）を選択するものだ。例えば次のような例があるとする。（WHERE 句を併用する例は、HACK 8.5 参照）

■SQL

```
-- 例1. 表"MY_TBL"から、"id"列、"name"列を、この順番に選択
SELECT "id", "name" FROM "MY_TBL";

-- 例2. 表"MY_TBL"の中の列全部を元の順番で選択
SELECT * FROM "MY_TBL";

-- 例3. 表"MY_TBL"から、"id"列を選択し、2列目に即値"10"を付ける
SELECT "id",10 FROM "MY_TBL";
```

これはちょうど、AWK コマンドの print ステートメントに相当する。

■UNIX(1) – AWK コマンド使用

```
# 例1. 表ファイル"MY_TBL.txt"から、1列目、2列目を、この順番に選択
```

```
awk '{print $1,$2;}' "MY_TBL.txt"

# 例2. 表ファイル"MY_TBL.txt"の中のすべての列を元の順番で選択
awk '{print;}' "MY_TBL.txt"

# 例3. 表ファイル"MY_TBL.txt"から、1列目を選択し、2列目に即値"10"を付ける
awk '{print $1,10;}' "MY_TBL.txt"
```

7.1.1 列名でなく列番号で指定するという文化の違いに注意

ただし、置き換えをするうえで大きな問題がある。それは、SELECT 文を含む SQL の世界では、列は名前（列名）で指定していたのに対し、**UNIX の世界では番号（列番号）で指定しなければならない**という違いがあることだ。

従って、何番目の列は何のデータが入っているかは常に指定しなければならない。それに、設計変更によって列数が変更されると、変更された列以降の番号がずれるため、番号の更新漏れがあると参照データが狂ってしまうというという危険性がある。

この性質だけ見れば、UNIX ファイルシステム上での実装は SQL での実装よりも劣るように見えるが、ただその一点だけをもって全否定するようでは、使う側の資質に問題があると言わざるを得ない。いかに上手に付き合うかを考えてみるのが先だ。

なお、例 3 のような「即値を挿入する」という必要がなければ、Tukubai コマンドの self（“SELect Fields” の意味）[*1] を使うと、もっと簡単に書ける。

■UNIX(2) – self コマンド使用

```
# 例1. 表ファイル"MY_TBL.txt"の中から、1列目、2列目を選択
self 1 2 "MY_TBL.txt"

# 例2. 表ファイル"MY_TBL.txt"の中のすべての列を選択
self 1/NF "MY_TBL.txt"
```

補足しておくと、例 2 に出てくる “NF” は AWK の同名組み込み変数と同じ意味であり、つまり “1/NF” とは「1 列目から最終列」までという意味である。また、ファイル名（self コマンドの最後に来る）が単純な数字であるなどして、列番号としての引数と区別がつかないと誤動作するのでこの点にも注意が必要[*2]。

このようにして、列番号の移動はこのコマンドを使えば簡単に移動できるのだから、上手に活用して番号で列を指定するというハンディーを上手に付き合う方法を考えるべきだ。

HACK 7.2 INSERT コマンド

INSERT コマンドは、表に対して行を追加するものである。

[*1] 詳細は次の URL の man ページ参照。
https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_self

[*2] もしファイル名が単純な数字であるなら “./1” のようにして、パス名を含めること。

例えば次のような例があるとする。

■SQL

```
-- 例1. 表"MY_TBL"に、("id","name")の列順で(1,'John')と(2,'Alice')の2行を追加
INSERT INTO "MY_TBL" ("id", "name") VALUES (1,'John'), (2,'Alice');

-- 例2. 表"MY_TBL2"に、表"MY_TBL1"の内容を全て追加
INSERT INTO "MY_TBL2" SELECT * FROM "MY_TBL1";
```

HACK 3.1 で指摘したように、RDB の表 1 つをファイル 1 つに置き換えることは必ずしも適当ではない（大きな表ならそれを複数のファイルで管理する方がよい）ことに気を付けなければが、それがちゃんと考慮されているという前提で話を進めると、表に行を追加するということは、ファイルに行を追加書き込みすることに相当するので、ファイルリダイレクション（追記）が使える。この場合、SQL の INSERT コマンドに直接対応するのは何らかの UNIX コマンドではなく、シェルの追記リダイレクション記号 “>>” である。

■UNIX(1) – 追記リダイレクション（単調増加データ向け）

```
# 例1. 表ファイル"MY_TBL.txt"に、("id","name")の列順で
#      (1,'John')と(2,'Alice')の2行を追加
echo '1 John'  >> "MY_TBL.txt"
echo '2 Alice' >> "MY_TBL.txt"

# 例2. 表ファイル"MY_TBL2.txt"に、
#      表ファイル"MY_TBL1.txt"の内容を全て追加
cat "MY_TBL1.txt" >> "MY_TBL2.txt"
```

ただし、例えば表ファイルには列が 3 つあるのに 2 つの列のデータしか与えないとか、表ファイルでは「ID」、「名前」、「生年月日」の順に格納されているのに、そこに「名前」、「ID」、「生年月日」のように異なる列順でデータを与えてはいけない。SQL の INSERT コマンドのように、適宜 NULL を挿入してくれたり、列順を認識はしてくれないからだ。

また、この追記リダイレクションによる追加は、タイムスタンプ順（単調増加する）にデータを追記していくログデータのようなものには向いている（HACK 4.1 参照）ものの、50 音順にソートすることが求められる名簿のようなデータに対して行うとソート順が乱れてしまう。その場合には次の項のようなやり方をするのがよい。

7.2.1 ソート順を維持しながら追記する

3.3.4 の「更新するとソート処理が発生するが、その考慮は?」項でも指摘したが、表ファイルは通常、ソートが済んでいる状態を維持しておいた方が計算量を節約できる（つまり速い）。しかし、追記リダイレクション “>>” を行うと、大抵の場合ソート順序が乱れてしまう。

これを防ぐには、同 HACK で示したように UNIX の sort コマンドの “-m” オプションを用いてマージソートを行う。

■UNIX(2) – sort コマンド使用

```
# 例1. 表ファイル"MY_TBL.txt"に、("id","name")の列順で
```

```
#       (1,'John')と(2,'Alice')の2行を追加し、
#       新たな表ファイル"MY_TBL2.txt"に書き込む
#       ただし、1列目の"id"順(文字順)でのソートを維持
{
  echo '1 John'
  echo '2 Alice'
  # ※ echoで書き込む行の順番は、ソート済の状態であること
}                                |
sort -bm -k 1,1 "MY_TBL.txt" - > "MY_TBL2.txt"

# 例2. 表ファイル"MY_TBL2.txt"に、
#       表ファイル"MY_TBL1.txt"の内容を全て追加し、
#       新たな表ファイル"MY_TBL3.txt"に書き込む
#       ただし、1列目の文字順でのソートを維持
cat  "MY_TBL1.txt" |
sort -bm -k 1,1 "MY_TBL2.txt" - > "MY_TBL3.txt"

# 例2. (別解)
sort -bm -k 1,1 "MY_TBL2.txt" "MY_TBL1.txt" > "MY_TBL3.txt"

# ※ "MY_TBL1.txt"も"MY_TBL2.txt"も、1列目によるソート済であること
```

使用上の注意としては、未ソートであるならソートを済ませておかなければならない。マージソートはその性質上、ソート前の各表ファイルについてはソート済でなければならないからだ。

なお、元の表ファイルに書き戻していないが、その理由は HACK 4.5 で説明した通り、File/Dir Hack においてはデメリットが多いからである。

ところで、データによっては安定ソートが必要な場合がある。安定ソートとは、ソート指定されていない列については元の順番を維持するというものだ。例えば、

```
     :
2021-12-06 みやもと
2021-12-07 いかわ
2021-12-07 いいやま
2021-12-08 なかやま
     :
```

のようにサインアップ日とサインアップ者の名前から構成された表があったとして、これをサインアップ日について降順に並べ替えたいとする。ただし、何らかの事情があって同じ日に登録された名前の順番は元の順番を維持してもらいたいという要求がなされた時、これに応えるのが安定ソートである。一部の sort コマンドは “-s” オプションによってこれに対応しており、上記の例であれば “`sort -bs -k 1r,1`” と記述した sort コマンドに流し込むことでで き、次のように並ぶ。(元の順番通り、「いかわ」が「いいやま」の前にある)

```
     :
2021-12-08 なかやま
2021-12-07 いかわ
```

```
2021-12-07 いいやま
2021-12-06 みやもと
     :
```

7.2.2　ソート順を維持しながら追記する（安定ソートが必要な場合）

しかし、sort コマンドの “-s” オプションは POSIX 標準ではサポートされていないので POSIX 原理主義に反する。そこで、Tukubai コマンドの一つである up3 コマンド[*3]を使う。ShellShoccar コマンドセットとして収録した up3 コマンドは “-s” オプション対応の sort コマンドが存在する場合には内部でそれを利用するが、存在しない場合には POSIX の範囲の AWK で実装した安定ソートを行うように作ってある。

■UNIX(3) – up3 コマンド使用（安定ソートを要するデータ向け）

```
# 例1. 表ファイル"MY_TBL.txt"に、("id","name")の列順で
#      (1,'John')と(2,'Alice')の2行を追加し、
#      新たな表ファイル"MY_TBL2.txt"に書き込む
#      ただし、1列目の"id"順(文字順)でのソートを維持
{
  echo '1 John'
  echo '2 Alice'
  # ※ echoで書き込む行の順番は、ソート済の状態であること
}                          |
up3 key=1 "MY_TBL.txt" - > "MY_TBL2.txt"

# 例2. 表ファイル"MY_TBL2.txt"に、
#      表ファイル"MY_TBL1.txt"の内容を全て追加し、
#      新たな表ファイル"MY_TBL3.txt"に書き込む
#      ただし、1列目の文字順でのソートを維持
cat       "MY_TBL1.txt"   |
up3 key=1 "MY_TBL2.txt" - > "MY_TBL3.txt"

# 例2.（別解）
up3 key=1 "MY_TBL2.txt" "MY_TBL1.txt" > "MY_TBL3.txt"

# ※ "MY_TBL1.txt"も"MY_TBL2.txt"も、1列目によるソート済であること
```

up3 コマンドも内部的にはマージソートを行っているので、このコマンドに与える表ファイル（データ）はすべてソート済でなければならない。

[*3] https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_up3

HACK 7.3 UPDATE コマンド

UPDATE コマンドは、既存表の中の特定の列の値を更新するものであるが、まずはその例を示す。（WHERE 句を併用する例は、HACK 8.5 参照）

■SQL

```
-- 例．表"MY_TBL"で、すべての行の"Z"列について、"X"列の10倍+"y"列の値を代入
UPDATE "MY_TBL" SET "Z" = "X" * 10 + "Y";
```

各列の値を加工するといったら、UNIX の世界では AWK コマンドである。従って、上記の例は次のように置き換えられる。

■UNIX

```
# 例．表ファイル"MY_TBL.txt"で、すべての行の3列目について、
#     1列目の10倍+2列目の値を代入し、新たな表ファイル"MY_TBL2.txt"に書き込む
#     （列数はわからない、または、何列であっても対応できる書き方）
awk '{$3=$1*10+$2; print;}' "MY_TBL.txt" > "MY_TBL2.txt"

# 上記の表ファイル"MY_TBL.txt"の列数がわかっていてる場合の効率的な書き方
# （5列であった場合）
awk '{print $1, $2, $1*10+$2, $4, $5;}' "MY_TBL.txt" > "MY_TBL2.txt"
```

注意しなければならないのは、SQL 文の UPDATE コマンドのように更新の必要な列だけを書けば良いわけではない。存在するすべての行が書き出されるように、AWK の print ステートメントに全列の変数を書くか、“print” とだけ書く（この場合は行全体の意味になる）ようにする。

最初の AWK 文の例のように書けば最後の print ステートメントで全列が書き出されるので一応これで問題無いのだが、AWK は列データが入っている変数 “$*n*” に値を再代入すると行全体の文字列 “$0” が再生成する処理が発生して速度が低下してしまう。これを回避するためには、例えば 2 番目の AWK 文のように書く。ただし、この書き方は列数が不明な場合には使えない。

HACK 7.4 MERGE コマンド

MERGE コマンドは、2 つの表 A,B を比較し、定めた条件に合致する行が見つかった、または見つからなかったかに応じて、INSERT コマンドか UPDATE コマンドを実行するというものである。

典型的な使い方としては、何らかのマスター表 TBL_MST があって、それを更新情報が格納された表 TBL_DIFF に基づいて更新したいという場合に用いられる。

具体的には次のような使い方である。

- TBL_DIFF の主キー列の値で、TBL_MST には同じ値を持つ行がない場合
  → TBL_DIFF の当該行を TBL_MST に INSERT する
- TBL_DIFF の主キー列の値で、TBL_MST には同じ値を持つ行があった場合
  → TBL_DIFF の当該行で TBL_MST の当該行を UPDATE する
- TBL_MST のその他の行
  → そのまま

さらに具体例を示すなら、例えば TBL_MST と TBL_DIFF がそれぞれ次のような構成だった場合に、

ID（主キー）	駅名	注釈
:	:	:
JY25	品川	特になし
JY27	町田	特になし
:	:	:

ID（主キー）	駅名	注釈
JY26	高輪ゲートウェイ	2020/03/14 開業
JY27	田町	誤字を修正

マスター表（TBL_MST）を次の内容に更新する操作である。

ID（主キー）	駅名	注釈
:	:	:
JY25	品川	特になし
JY26	高輪ゲートウェイ	2020/03/14 開業
JY27	田町	誤字を修正
:	:	:

この操作は SQL では次のように記述できる。

■SQL

```
-- 例．マスター表"TBL_MST"を更新情報表"TBL_DIFF"の内容で更新（MERGE）
--      （両表の主キー列は"ID"）
MERGE INTO "TBL_A" AS m USING "TLB_DELTA" AS d ON m."ID" = d."ID"
  WHEN     MATCHED THEN
    UPDATE SET m."駅名" = d."駅名", m."注釈" = d."注釈"
  WHEN NOT MATCHED THEN
    INSERT ("ID", "駅名", "注釈") VALUES (d."ID", d."駅名", d."注釈");
```

UNIX の世界では、安定ソート（“-s” オプション）対応 sort コマンドと AWK コマンドを使えば書けるが、Tukubai コマンドの up3 と getlast を使うと簡単に書ける。

■UNIX(1) – up3,getlast 使用

```
# 例．マスター表ファイル"TBL_MST.txt"の内容を、
#     更新情報表ファイル"TLB_DIFF.txt"の内容で更新（MERGE）し、
#     新たな表ファイル"TBL_MST2.txt"に書き込む。
#      （主キー列は1列目）
cat "TLB_DIFF.txt"        |
up3 key=1 "TBL_MST.txt" - |
getlast 1 1               > "TBL_MST2.txt"

# ※ "TBL_MST.txt"も"TLB_DIFF.txt"も、1列目によるソート済であること
```

```
#     （up3もgetlastもソート済であることを要求する）
```

up3 は 7.2.2 の中の「ソート順を維持しながら追記する（安定ソートが必要な場合）」項でも説明したように、特定の列（a とする）のソート順を維持しながら 1 つの表（A とする）にもう 1 つの表（B とする）の行を挿入していくコマンドである。（ただし、挿入されてできた表は標準出力に送られるためめ別の表ファイルになる）だが、その列 a に関して、表 A に既に存在する値が表 B にもあった場合、表 B の当該行は表 A の当該行の次に挿入される。

先程の TBL_MST を表 A、TBL_DIFF を表 B、そして “ID” を列 a として説明するなら、up3 コマンド実行直後の表データは次のようになっている。

ID（主キー）	駅名	注釈
:	:	:
JY25	品川	特になし
JY26	高輪ゲートウェイ	2020/03/14 開業
JY27	町田	特になし
JY27	田町	誤字を修正
:	:	:

列 a に “JY27” を持つ行が 2 つ存在しているが、下にあるのが表 B 由来のものである。getlast コマンドは、このような一意制約に反する状態を解消するために利用できる。“getlast 1 1” とは、「1 行目から 1 行目（この場合は結局 1 行目）を主キーと見なし、重複する場合は最後の行のみ残すという働きをする。[*4]

結果として、表データは次のようになる。

ID（主キー）	駅名	注釈
:	:	:
JY25	品川	特になし
JY26	高輪ゲートウェイ	2020/03/14 開業
JY27	田町	誤字を修正
:	:	:

なお、getlast コマンドも引数で指定した範囲（いわゆる主キー扱いの列）の列によってソート済のデータを要求する。当該列の値が変わる手前の行を選び出すようなアルゴリズムになっているからである。

HACK 7.5 DELETE コマンド

DELETE コマンドは、既存表の中の特定の行を削除するものである。このコマンドは通常 WHERE 句と一緒に使うので、そのような例を示す。

■SQL

```
-- 例．表"MY_TBL"で、"id"列が100以上、150未満の行を削除
DELETE FROM "MY_TBL" WHERE "id" >= 100 AND "id" < 150;
```

UNIX の世界では、特定の列の条件に応じて何かさせるといったら AWK コマンドの出番である。条件

[*4] 詳細は次を参照。https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_getlast

に合致した場合は何もしないということであれば AWK では next ステートメント（その行の処理を中止する）を使うとよいが、合致しなかった場合はその行を書き出さなければならないので、結局元の DELETE 文は次のように書き換えられる。

■UNIX(1) – AWK コマンド使用

```
# 例．表ファイル"MY_TBL.txt"で、1列目が100以上、150未満の行を削除して、
#     新たな表ファイル"MY_TBL2.txt"に書き込む
awk '$1>=100 && $1<150{next;}; {print;}' "MY_TBL.txt" > "MY_TBL2.txt"
```

行を消すかどうかを行全体の文字パターンで判定できる場合は、grep コマンドや sed コマンドを使うこともできる。これらが使える場合は、こちらを使った方が動作が速かったり簡潔に書ける場合が多い。

■UNIX(2) – grep, sed コマンド使用

```
# 上記のUNIX(1)の例をgrepコマンドで書いた場合
# ・1列目の数字が100以上150未満であるなら、正規表現が使える
# ・"-v"オプションを使うと、条件に該当した行のみ出力しないという動作になる
grep -v '^1[0-4][0-9] ' "MY_TBL.txt" > "MY_TBL2.txt"

# 上記のUNIX(1)の例をsedコマンドで書いた場合
# ・同じ正規表現が使える
# ・該当した行を消す場合、sedでは"d"コマンドを使う
sed '/^1[0-4][0-9] /d' "MY_TBL.txt" > "MY_TBL2.txt"
```

HACK 7.6　TRUNCATE コマンド

TRUNCATE コマンドは、結果的には DELETE コマンドを WHERE 句なしで実行するのと同じであり、つまり表の中の行がすべて削除されるが、それが高速に行われる点が DELETE とは異なる。

■SQL

```
-- 例．表"MY_TBL"内のすべての行を削除
TRUNCATE TABLE "MY_TBL";
```

UNIX の世界でも表ファイルの中身を高速に削除する方法がある。最も速いのは “:” というコマンドを使う方法であろう。

■UNIX

```
# 例．表ファイル"MY_TBL.txt"内のすべての行を削除
: > "MY_TBL.txt"
```

“:” は「何もしない」（何も出力しない）コマンドであり、これを目的のファイルに上書きリダイレクト “>” すれば瞬時に 0 バイトファイルになる。

HACK 7.7 DROP コマンド

DROP コマンドは既存表そのものを削除するコマンドであり、典型的には “DROP TABLE” という句の後に削除したい表名を書く。(複数書ける実装もある)

■SQL

```
-- 例1. 表"MY_TBL"を削除(存在するとわかっている場合)
DROP TABLE "MY_TBL";

-- 例2. 表"MY_TBL1"と"MY_TBL2"を削除(複数表をまとめて指定)
DROP TABLE "MY_TBL1","MY_TBL2";

-- 例3. 表"MY_TBL"を削除(存在しない可能性がある場合)
DROP TABLE IF EXISTS "MY_TBL";
```

UNIX の世界では表はファイルとして存在しているので、すなわち rm コマンドを使えばよい。

■UNIX – rm コマンド使用

```
# 例1. 表ファイル"MY_TBL.txt"を削除(存在するとわかっている場合)
rm "MY_TBL.txt"

# 例2. 表ファイル"MY_TBL.txt"と"MY_TBL2.txt"を削除(複数ファイルをまとめて指定)
rm "MY_TBL1.txt" "MY_TBL2.txt"

# 例3. 表ファイル"MY_TBL.txt"を削除(存在しない可能性がある場合)
# ・"-f"オプションを付ければ削除できなくてもエラーを返さない
rm -f "MY_TBL.txt"
```

HACK 7.8 CREATE / ALTER コマンド

CREATE コマンドは表の構造を定義して新規作成するコマンド、一方 ALTER は既存表の構造の定義を変更するコマンドであるが、UNIX の世界にはこれらに相当するコマンドはない。なぜなら、File/Dir Hack で表として利用している「ファイル」という存在は単なるバイト列であって、それには列構造とか型などといった概念が無いからである。

ただし、SQL の表名を付ける、変更するという操作に限るなら、UNIX では “:”(SQL の TRUNCATE に相当する操作として紹介した HACK 7.6 のものと同じ)や、ファイル名変更に用いられる mv コマンドが相当する。

■SQL

```
-- 例1. 表"MY_TBL"を作成
CREATE TABLE "MY_TBL" ( ここに列定義が入る );
```

```
-- 例2. 表"MY_TBL1"という名前を"MY_TBL2"に変更
ALTER TABLE "MY_TBL1" TO "MY_TBL2";
```

■UNIX

```
# 例1. 表ファイル"MY_TBL.txt"を新規作成
: > "MY_TBL.txt"

# 例2. 表ファイル"MY_TBL1.txt"を"MY_TBL2.txt"に名前変更
mv "MY_TBL1.txt" "MY_TBL2.txt"
```

第 8 章

SQL to UNIX マイグレーション (2) – 選択の句

前章に引き続き、SQL の各「句」を UNIX の世界に翻訳（=UNIX コマンドで実装）する例を示していく。

この章で取り扱うのは、FROM、WHERE といった、表・行・列を選んだり、あるいは選ばれたそれら集合を並べ替える句とする。「選択」とは呼べない句は次章で、関数と呼べるもの（句の直後に丸括弧で引数を取るもの）は、その次の章で扱う。

そして引き続きこの章でも POSIX 標準の UNIX コマンドのみならず、HACK 1.7 で紹介した “Shell-Shoccar” コマンドセット（Tukubai コマンド含む）が登場するので、その HACK を参照して準備しておくこと。

HACK 8.1　FROM 句

FROM 句は、SELECT コマンドや DELETE コマンドの構文の中で出てくる句である。SELECT コマンドにおいては、これからデータを抽出しようとする対象となる表の指定に用いられ、DELETE コマンドにおいては、これからデータ（この場合は行）を選択的に削除するので、その条件判定（および削除）のために開かれる表の指定に用いられる。

UNIX の世界において表はファイルとして構成されているわけだが、そのファイルの中身を参照するといったら真っ先に候補に挙がるのは cat コマンドである。他にもいくつか方法があるが、まとめると次のとおりである。

a. cat コマンドで開く方法
   - その後で用いる予定のほとんどのコマンド（標準入力受入可能なもの）に有効

b. cat を用いず、その後で用いる予定のコマンドに入力リダイレクション “<” で与える方法
   - これも、ほとんどのコマンド（標準入力受入可能なもの）に有効

c. 開いた後で用いる予定のコマンドに引数として与える方法
   - AWK, sed など多くのコマンドが対応している。
   - 対応していない数少ないコマンドとしては tr がある。

具体例については、SELECT や DELETE コマンドについて記した HACK 7.1 や HACK 7.5 を参照。

HACK 8.2 UNION、UNION ALL 句

UNION と UNION ALL 句は、2 つまたはそれ以上の表（列構成が同じもの）それぞれが持つ行を合わせて、1 つの表として扱えるようにするためのものである。両者の違いは、合わせた後にまったく同一の行が見つかった場合にそれを除去するか（UNION）しないか（UNION ALL）の違いである。

例えば会員情報の表があるが、1 つではなく、男性の表 “MEMBERS_M” と女性の表 “MEMBERS_W”、どちらも好まない人の表 “MEMBERS_X” の 3 つあったとする。それらに対して “MEMBERS_M UNION MEMBERS_W UNION MEMBERS_X” と書けば、すべての性別の人をまとめた 1 つの表ができあがる。

SQL 文の例を挙げるとこうなる。

■SQL

```
-- 例1. 各性に分かれている会員表の各行を合わせて出力する（重複チェック有）
SELECT * FROM MEMBERS_M UNION MEMBERS_W UNION MEMBERS_X;

-- 例2. 各性に分かれている会員表の各行を合わせて出力する（重複チェック無）
SELECT * FROM MEMBERS_M UNION ALL MEMBERS_W UNION ALL MEMBERS_X;

-- 例3. 各性に分かれている会員表に性別列"sex"が無く、合わせる際に性別列"sex"を
--      設ける場合（この場合は重複チェック不要）
SELECT * FROM (SELECT *, 'M' AS "sex" FROM "MEMBERS_M")
               UNION ALL
              (SELECT *, 'F' AS "sex" FROM "MEMBERS_W")
               UNION ALL
              (SELECT *, 'X' AS "sex" FROM "MEMBERS_X");
```

UNIX の世界では多くの場合、最初に cat コマンドを使うことになる。複数のファイルを受け付けるコマンドであれば cat コマンドを使わなくてもできるが、それはコマンド個別の仕様に基づく。また、HACK 8.1 で記した入力リダイレクションは使えない。

■UNIX

```
# 例1. 各性に分かれている会員表ファイルの各行を合わせて出力する（重複チェック有）
cat "MEMBERS_M.txt" "MEMBERS_W.txt" "MEMBERS_C.txt" |
sed 's/[[:blank:]]\{2,\}/ /g'                        | #←列区切りの半角空白数が
sort                                                 | #          不定の時必要
uniq

# 例2. 各性に分かれている会員表ファイルの各行を合わせて出力する（重複チェック無）
cat "MEMBERS_M.txt" "MEMBERS_W.txt" "MEMBERS_C.txt"

# 例3. 各性に分かれている会員表ファイルに性別列"sex"が無く、
#      合わせる際に性別列"sex"を1列目に設ける場合（この場合は重複チェック不要）
(     awk '{print "M",$0;}' "MEMBERS_M.txt") |
```

```
(cat; awk '{print "F",$0;}' "MEMBERS_W.txt") |
(cat; awk '{print "X",$0;}' "MEMBERS_X.txt")
```

重複チェックが必要な場合、UNIX の世界では複数の表ファイルを出力した後、sort コマンドと uniq コマンドを合わせた “sort | uniq” を通す。または “sort -u” のように “-u” オプションを使って sort コマンド 1 つにしてもいい。これは暗記してもいいほどに定番の使い方だ。ただし、列区切りの半角空白の数が各表でバラバラである恐れがある場合には sort の手前に例示したような sed コマンドを挿入し、予め半角空白を 1 つに揃えておくこと。

重複チェック不要な場合は、cat コマンドで複数の表ファイルを出力するだけでいい。cat コマンドとはそもそも concatinate（結合する）という意味からきていて、cat コマンド本来の使い方と言われている。なお、cat コマンドは単に行を合わせるだけなので、順番を揃えたければ（ORDER BY したければ）その後に sort コマンドを繋げることになる。（HACK 9.1 参照）

最後の例は、行を合わせる際に列を追加するという応用例である。各表への列の追加は AWK コマンドでできるし、その後に行を合わせるにはサブシェル（丸括弧）と cat コマンドを組み合わせて、(cat; ここに任意のコマンド) とやればいい。このテクニックを使えば、標準入力から到来したデータを出力した後で新たなデータを後ろに追加できる（cat を後ろに置けば、新たなデータの方を先頭に追加できる）。なお、例 3 の 1 行目に丸括弧を置いているのは見た目のためであり、無くても構わない。

HACK 8.3　EXCEPT / MINUS 句

EXCEPT 句または MINUS 句（製品によって名前が異なる）は UNION 句に類似した句の一つであり、UNION 句が和集合を得るためのものであるのに対して、EXCEPT 句は差集合を得るためのものである。例えば先月時点の会員表 “MEMBERS” と、今月退会した人の表 “WITHDRAWALS” があったとして、今月時点の会員表が欲しいという場合に使える。

というわけでこの例を SQL 文で表現すると次のようになる。

■SQL

```
-- 例．会員表"MEMBERS"から、退会者表"WITHDRAWALS"にある行を除いた行だけ出力
SELECT * FROM "MEMBERS" EXCEPT "WITHDRAWALS";
```

UNIX の世界には、これとほぼ一致した動作をしてくれるコマンドとして comm というものがある。

■UNIX

```
# 例．会員表ファイル"MEMBERS.txt"から、退会者表ファイル"WITHDRAWALS.txt"にある
#     行を除いた行だけ出力
#     ※ 両ファイルとも文字列ソート済かつ、列区切りを半角空白1文字に統一しておくこと
comm -23 "MEMBERS.txt" "WITHDRAWALS.txt"

# （両ファイルとも未ソートだったり列区切りが統一されていない場合）
cat "MEMBERS.txt"             |
sed 's/[[:blank:]]\{2,\}/ /g' |
sort                          > "$Tmp/MEMBERS.normalized.txt"
cat "WITHDRAWALS.txt"                       |
sed 's/[[:blank:]]\{2,\}/ /g'               |
```

```
sort                                    |
comm -23 "$Tmp/MEMBERS.normalized.txt" -
rm "$Tmp/MEMBERS.normalized.txt"
```

comm コマンドとは、引数から与えられた 2 つのファイルを比較し、1 番目のファイルにのみ存在する行、2 番目のファイルにのみ存在する行、両方に存在する行を仕分けする。その際、1 から 3 までの数字オプションを取ることができ、それぞれ次の意味を持っている。

- `-1` 引数の 1 番目で指定されたファイルにのみ存在する行は表示しない。
- `-2` 引数の 2 番目で指定されたファイルにのみ存在する行は表示しない。
- `-3` 両方のファイルのどちらにも存在する行は表示しない。

従って、“`-2`” と “`-3`” の両方を指定すれば、1 番目のファイル（つまり “`MEMBERS.txt`”）に存在し、かつ 2 番目のファイル（つまり “`WITHDRAWALS.txt`”）に存在しない行のみ出力することができる。

なお、comm コマンドもまた事前にソートが必要なものの 1 つであるので、HACK 2.10 以降で言ってきたようにソート済の状態で保存しておくべきである。両方のファイルが未ソートだったり列区切りが統一されていない場合は、後者の例のようにして一時ファイルを経由しなければならない。（一時ファイルに関しては HACK 2.11 参照）

さらに補足として、EXCEPT 句とは直接関係ないことだが、comm コマンドに数字オプションを 0 個または 1 個しか指定しなかった場合は行頭に 0〜2 個のタブ文字（0x09）が挿入され、どちらのファイル由来（あるいは両方のファイル由来）なのかが区別できるようにされているので注意。

HACK 8.4 INTERSECT 句

INTERSECT 句もまた UNION 句や EXCEPT 句に類似した句の一つであり、UNION 句が和集合、EXCEPT 句が差集合を得るものであるのに対し、INTERSECT 句は重複した行のみから成る集合を得るものである。

例えば、2010 年時点の会員表 “`MEMBERS_2010`” と、2020 年時点の会員表 “`MEMBERS_2020`” があったとして、どちらの年にも在籍していた会員のみを抽出したいという場合に使える。

というわけでこの例を SQL 文で表現すると次のようになる。

■SQL

```
-- 例．会員表の2010年版"MEMBERS_2010"と2020年版"MEMBERS_2020"の
--       どちらにも在籍していた会員の行だけ出力
SELECT * FROM "MEMBERS_2010" INTERSECT "MEMBERS_2020";
```

UNIX の世界では、HACK 8.3 でも紹介した comm コマンドを使えばほぼ同様に処理できる。

■UNIX

```
# 例．会員表ファイルの2010年版"MEMBERS_2010.txt"と、2020年版"MEMBERS_2020.txt"の
#       どちらにも在籍していた会員の行だけ出力
#       ※ 両ファイルとも文字列ソート済かつ、列区切りを半角空白1文字に統一しておくこと
comm -12 "MEMBERS_2010.txt" "MEMBERS_2020.txt"

# （両ファイルとも未ソートだったり列区切りが統一されていない場合）
```

```
cat "MEMBERS_2010.txt"         |
sed 's/[[:blank:]]\{2,\}/ /g' |
sort                           > "$Tmp/MEMBERS_2010.normalized.txt"
cat "MEMBERS_2020.txt"                              |
sed 's/[[:blank:]]\{2,\}/ /g'                       |
sort                                                |
comm -12 "$Tmp/MEMBERS_2010.normalized.txt" -
rm "$Tmp/MEMBERS_2010.normalized.txt"
```

EXCEPT 句の時との違いは、comm コマンドのオプションを “-2” と “-3” ではなく “-1” と “-2” にするだけである。であるから、comm コマンドに関するそれ以上の説明は HACK 8.3 を参照。

HACK 8.5　WHERE 句

WHERE 句は SELECT（選択）・UPDATE（更新）・DELETE（削除）の各コマンドで、選択・更新・削除の対象とする行の条件を指定するためのものである。従って、WHERE 句の後には条件式を置く。

例えば次のように書ける。（例に登場する LIKE 句については、HACK 8.9 を参照）

■SQL

```
-- 例1. 会員表"MEMBERS"から、年齢列"age"が20歳代の行を出力
SELECT * FROM "MEMBERS" WHERE "age" >= 20 AND "age" < 30;

-- 例2. 駅情報表"STATIONS"の"ID"が"TR03"の行の"ローマ字"を"Hasama"に修正
UPDATE "STATIONS" SET "ローマ字" = 'Hasama' WHERE "ID" = 'TR03';

-- 例3. 過去メッセージ格納表"IM_LOG"で、表に存在する列のいずれか（※）に
--      文字化けで生じる下駄記号が"＝"が含まれる行をすべて削除
--      ※ SQLではSELECTの"*"以外、存在する列すべてを指定する方法が見当たらず、
--         すべての列を列挙するしかなさそう。
DELETE FROM "IM_LOG" WHERE 列a || 列b || ...存在するすべての列) LIKE '%＝%';
```

UNIX の世界で、条件を指定して行を選ぶといえば AWK コマンドであり、基本的には AWK コマンドで WHERE 句に相当する操作ができる。ただし、上記の例 3 のように、どの列にあっても関係ないという場合（あるいは対象が 1 列目や最終列にあって正規表現で簡単に条件が記述できる場合）には grep コマンドが使える。

というわけで UNIX の世界に翻訳すると次のようになる。

■UNIX

```
# 例1. 会員表ファイル"MEMBERS.txt"から、年齢列（5列目とする）が20代の行を出力
awk '$5>=20 && $5<30{print;}' "MEMBERS.txt"

# 例2. 駅情報表ファイル"STATIONS.txt"のID列（1列目とする）が"TR03"の行の
#      ローマ字列（4列目とする）を"Hasama"に修正し、"STATIONS2.txt"に書き込む
awk '$1=="TR03"{$4="Hasama";}; {print;}' "STATIONS.txt" > "STATIONS2.txt"
```

```
# 例3. 過去メッセージ格納表ファイル"IM_LOG.txt"で、表に存在する列のいずれか（※）に
#      文字化けで生じる下駄記号が"＝"が含まれる行をすべて削除し、
#      "IM_LOG2.txt"に書き込む
#      ※ UNIXの世界では「すべての列・いずれかの列」を指す場合、
#         awkコマンドの"$0"も使えるし、grepコマンドも使える。
grep -vF '＝' "IM_LOG.txt" > "IM_LOG2.txt"

# （例3の別解）
awk '$0!~/＝/' "IM_LOG.txt" > "IM_LOG2.txt"
```

grep コマンドのオプションについて補足すると、“-v” オプションは条件に該当するものを逆に出力しないためのもので、“-F” はパターン文字列を正規表現と見なさず、単純な部分一致のためのパターンと解釈させるためのものである。

8.5.1 日時の大小比較はどう翻訳すべきか

WHERE 句に対応するものとして示した AWK コマンドには数値型、文字列型という二種類の型が暗示的に存在するだけで、日付型というものが存在せず、大小比較は数値と文字列（文字コード列）の比較にしか対応していない。

もし、日付データの文字数が統一されているのであれば文字列として大小比較しても正しい結果が得られる。例えば “2015/10/03” と “2015/07/03” というように、月日が 1 桁の場合でも 0 埋めしてある場合だ。しかし、日付データのある値が “2015/10/3” で、もう 1 つのある値が “2015/7/3” と記録されていた場合、これをそのまま比較すると前者が小さいと判定されてしまう。この問題を回避するための翻訳例を示す。

まずは、SQL 文例を記す。

■SQL

```
-- 例. 会員表"MEMBERS"から、入会年月日列"entdate"が2020年4月1日以降の人をすべて出力
SELECT * FROM "MEMBERS" WHERE "entdate" >= '2020-04-01';
```

これを UNIX の世界に翻訳する際、先程指摘したように日付の 0 埋めがなされていないデータでも正しく動くようにするには次のように書く必要がある。

■UNIX

```
# 例. 会員表ファイル"MEMBERS.txt"から、入会年月日列（11列目とする）が
#     2020年4月1日以降の人をすべて出力
cat "MEMBERS.txt"                        |
awk '{s=$11;                             #
      gsub(/[^0-9]/," ",s);              #
      split(s,a);                        #
      date=a[1]*10000+a[2]*100+a[3];     #
      if (date>=20100401) {print;}  }'
```

AWK の最初の 4 行で年月日を表す 8 桁の自然数（YYYYMMDD）を作っている。年月日の桁が揃っ

ていれば数値として問題なく比較できるというわけだ。もし時分秒まである日時を比較したければ、同様にして 14 桁の自然数（YYYYMMDDhhmmss）にすればよい。

もっとも、こういう面倒な処理を書かなくて済むように、日付データは桁揃えをした状態で格納しておくべきである。

HACK 8.6　GROUP BY 句

GROUP BY 句は、指定した条件に一致する行同士を集約するためのものである。例えば、会員表の性別列"sex"ごとに分けてそれぞれの平均年齢を求めたり、年齢列"age"を見て、10 代、20 代のように 10 歳単位で人数を集計する場合などに用いられる。

上記の例をより具体化させた SQL 文の例を示す。（平均値を意味する AVG 関数については HACK 10.5 を、切り捨てを意味する FLOOR 関数については HACK 10.18 参照）

■SQL

```
-- 例1. 会員表"MEMBERS"の性別列"sex"の値が同じ行同士を集計し、
--      それぞれの行にある年齢列"age"に基づいて平均年齢（小数点第2位で四捨五入）を
--      計算する
--      そして、"sex"を1列目、"aveage"を2列目に配置して出力
SELECT   "sex"                          ,
         ROUND(AVG("age"),1) AS "aveage"
FROM     "MEMBERS"
GROUP BY "sex";

-- 例2. 会員表"MEMBERS"の各行を、10歳刻みに要約した年齢列"age"と性別列"sex"から成る
--      グループ（「30代男性」のように）集約し、各グループの人数を集計する
--      集計後は、"generation"を1列目、"sex"を2列目、そのグループの"人数"を
--      3列目に配置して出力
SELECT   FLOOR("age"/10)*10 AS "generation",
         "sex"              AS "sex"       ,
         COUNT(*)           AS "population"
FROM     "MEMBERS"
GROUP BY FLOOR("age"/10), "sex";
```

これらを UNIX の世界に翻訳すると次のようになる。

■UNIX

```
# 例1. 会員表ファイル"MEMBERS.txt"の性別列（6列目とする）の値が
#      同じ行同士を集計し、それぞれの行にある年齢列（5列目とする）から
#      平均年齢（小数点第2位で四捨五入）を計算する
#      そして、性別を1列目、平均年齢を2列目に配置して出力
cat "MEMBERS.txt"                                                    |
awk '    {cnt[$6]++; sum[$6]+=$5;                                 } #
     END {for (i in cnt) {print i, int(sum[i]*10/cnt[i]+0.5)/10;}}'
```

```
# 例2. 会員表ファイル"MEMBERS.txt"の各行を、10歳刻みに要約した年齢列
#       （5列目とする）と性別列（6列目とする）から成るグループ
#       （「30代男性」など）に集約し、各グループの人数を集計する
#      集計後は、年代を1列目、性別を2列目、そのグループの人数を
#      3列目に配置して出力
#
#      ※ コマンドの間にあるコメント行（"1:年代 2:性別"など）は、
#         その位置を流れるデータの列構成を表している
#
cat "MEMBERS.txt"                |
awk '{print int($5/10)*10, $6;}' |
# 1:年代 2:性別                  #
sort -k 1n,1 -k 2,2              |
count 1 2
# 1:年代 2:性別 3:人数

# （例2をPOSIX標準コマンドのみで行う場合の別解）
cat "MEMBERS.txt"                |
awk '{print int($5/10)*10, $6;}' |
# 1:年代 2:性別                  #
sort -k 1n,1 -k 2,2              |
uniq -c                          |
# 1:人数 2:年代 3:性別           #
awk '{print $2,$3,$1;}'
# 1:年代 2:性別 3:人数
```

GROUP BY 句を翻訳するうえでのポイントはまず、GROUP BY 句の直後に記述している列名、あるいは演算式のとおりに演算をして、同一視すべきものが同一に見えるようにすることである。

例 1 では、同一視した文字列をキーにした連想配列を作り、年齢と人数を足しこんでいって最後に平均値を求めている。例 2 では、同一視すべき文字列（年代）と性別を並べた列（飛び飛びにせず隣同士に並べる）を標準出力し、次のコマンドで同一視すべき行が連続する回数を数えるのに利用している。

なお、例 2 では Tukubai コマンドセットの 1 つである count というコマンド[*1]を利用している。意味は、その第 1 引数から第 2 引数で指定された範囲の列を同一視して、同一な行が何回連続したかを数えよというものである。

count コマンドの動作を理解するための参考に、例 2 の別解をその次に記した。uniq コマンドの “-c” オプションは、同一行を集約するという uniq コマンドの基本動作に加えて、集約した行数を行頭に挿入するというものである。

[*1] 詳細は次の URL を参照。
https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_count

HACK 8.7　HAVING 句

HAVING 句は WHERE 句と同様に、条件式を示して行を選択するものである。ただし、WHERE 句が GROUP BY 句よりも前の段階で機能するのに対して、HAVING 句は後の段階で機能するという違いがある。

HACK 8.6 の例 2 を元に例題を考えると、例えば例 2 の結果に基づき、人数が 100 人以上のグループだけを出力するという場合には HAVING 句の出番である。

■SQL

```
-- 例．会員表"MEMBERS"の各行を、10歳刻みに要約した年齢列"age"と性別列"sex"から成る
--     グループ（「30代男性」のように）集約し、各グループの人数を集計する
--     集計後は、"generation"を1列目、"sex"を2列目、そのグループの"人数"を
--     3列目に配置して出力するものの、その際に100人未満の行に関しては出力しない
SELECT   FLOOR("age"/10)*10 AS "generation",
         "sex"              AS "sex"       ,
         COUNT(*)           AS "population"
FROM     "MEMBERS"
GROUP BY FLOOR("age"/10), "sex"
HAVING   "population" >= 100;
```

UNIX の世界では、HAVING 句に対しても WHERE 句と同様に AWK や grep コマンドが使える。HAVING 句が GROUP BY 句の後に作用するのだから、GROUP BY 句に相当するコマンドの後に AWK や grep をパイプで繋げばいいだけである。

■UNIX

```
# 例．会員表ファイル"MEMBERS.txt"の各行を、10歳刻みに要約した年齢列
#     （5列目とする）と性別列（6列目とする）から成るグループ
#     （「30代男性」など）に集約し、各グループの人数を集計する
#     集計後は、年代列を1列目、性別を2列目、そのグループの
#     人数列"population"を3列目に配置して出力
#     ただし、人数が100人未満の行は出力しない
#
#     ※ コマンドの間にあるコメント行（"1:年代 2:性別"など）は、
#        その位置を流れるデータの列構成を表している
#
cat "MEMBERS.txt"                |
awk '{print int($5/10)*10, $6;}' |
# 1:年代 2:性別                  #
sort -k 1n,1 -k 2,2              |
count 1 2                        |
# 1:年代 2:性別 3:人数           #
awk '$3>=100{print;}'
```

UNIX で実装する場合の利点は、データ処理の順番が上から下へ素直に記述できる点ではなかろうか。これは可読性向上に繋がる。SQL の場合、この例ではデータはまず FROM の行から始まって、GROUP BY の行の処理を受け、次に HAVING の行の処理を受け、最後は最初に戻って SELECT の行の処理を受ける。どうしてこうもいたずらにプログラマーの目線を弄ぶのか。副問い合わせ等が組み合わさって SQL 文が長くなる場合は目線はさらに上下に動かされることになる。

HACK 8.8 DISTINCT 句

DISTINCT 句は、指定列に同じ値を持つレコードが複数あった場合にそれを 1 つに集約するもので、GROUP BY 句と同じ目的にも使えることが多く、そして同じ結果を得られる場合が多い。

ただ、GROUP BY 句との違いは、SQL 文のどの段階で作用するかである。DISTINCT 句は、SELECT コマンドの直後に置いて直後に列名や演算式を指定するように記述することで、その結果に対して作用する。すなわち、得られた結果に重複が見つかった時点で集約をする。

実用的にはあまり意味の無い例ではあるが、DISTINCT 句の動作の再確認を兼ねて HACK 8.6 の SQL 文例に対して DISTINCT 句を追加する例を考えてみる。

■SQL

```
-- 例. 会員表"MEMBERS"の各行を、10歳刻みに要約した年齢列"age"と性別列"sex"から成る
--     グループ（「30代男性」のように）に集約し、各グループの人数を集計する
--     集計後は（年齢も性別なしで）単純にその人数のみを列挙し、
--     その際、同じ人数のものがあった場合には1つにまとめる
SELECT   DISTINCT COUNT(*) AS "population"
FROM     "MEMBERS"
GROUP BY FLOOR("age"/10), "sex";
```

これを UNIX の世界に翻訳すると次のようになる。

■UNIX

```
# 例. 会員表ファイル"MEMBERS.txt"の各行を、10歳刻みに要約した年齢列
#     （5列目とする）と性別列（6列目とする）から成るグループ
#     （「30代男性」など）に集約し、各グループの人数を集計する
#    集計後は（年齢も性別なしで）単純にその人数のみを列挙し、
#     その際、同じ人数のものがあった場合には1つにまとめる
#
#     ※ コマンドの間にあるコメント行（"1:年代 2:性別"など）は、
#        その位置を流れるデータの列構成を表している
#
cat "MEMBERS.txt"                |
awk '{print int($5/10)*10, $6;}' |
# 1:年代 2:性別                  #
sort -k 1n,1 -k 2,2              |
count 1 2                        #
# 1:年代 2:性別 3:人数           #
```

```
self 3                          |
# 1:人数                        #
uniq
```

UNIX の世界の記述例を見ることで、逆に SQL 文における DISTINCT 句の働きが理解できるのではないだろうか。こちらの例では、count コマンドの行まではまったく同じである。そして、SQL 文の方が DISTINCT 句で人数列のみに絞り込んでいるので、それに合わせて self コマンドを使って同様にして人数列のみに絞り込み、その後 DISTINCT 句の集約作用を再現するために uniq コマンドで集約している。

この例を見ても、UNIX の世界のワンライナーの方が作用の順序が上から下へ素直に記述できているので理解しやすいように思うが、どうだろう。

HACK 8.9 [NOT] (LIKE / REGEXP / SIMILAR TO) 句

LIKE 句を始めとしたこれらの句は、条件演算子の一種であり、列などにある文字列が句の直後に示したパターンに一種していれば真となるものである。(標準 SQL の) LIKE 句のパターンは UNIX でファイル名を指定する時のワイルドカードに似ていて、ワイルドカードの “?” と “*” は、それぞれ “_” と “%” に対応する。SQL Server の LIKE 句、MySQL の REGEXP 句、PostgreSQL の SIMILAR TO 句は、パターンとして正規表現を使える。Oracle では同様のものが REGEXP_LIKE という関数で提供されている。なお、これらの句の手前に NOT が付くと真偽が反転する。そして、演算子の一種であるので WHERE 句などと共に用いる。

SQL 文での使用例を示す。

■SQL

```
-- 例. 会員表"MEMBERS"に登録されている行のうち、
--      会員ID列"id"の値（アルファベット1文字+4桁の数字という規則）の
--      数字の1文字目が"5"である会員をすべて出力
SELECT   *
FROM     "MEMBERS"
WHERE    "id" LIKE '_5%';
```

UNIX の世界に翻訳すると、WHERE 句（HACK 8.5 参照）のところに記したように、AWK コマンドまたは grep コマンド（行全体でマッチングできる場合）が使える。ただし、AWK にも grep にも “_” や “%” の役割に相当する文字が無いため、正規表現を代用しなければならないが、こちらの方が表現力が上なので問題ないだろう。

■UNIX

```
# 例. 会員表ファイル"MEMBERS.txt"に登録されている行のうち、
#      会員ID列（1行目とする）の値（アルファベット1文字+4桁の数字という規則）の
#      数字の1文字目が"5"である会員をすべて出力
awk '$1~/^.1.+$/{print;}' "MEMBERS.txt"
```

なお、AWK と grep それぞれで使える正規表現メタ文字に関しては「正規表現メモ」というページ[*2]が大変参考になる。

[*2] http://www.kt.rim.or.jp/ kbk/regex/regex.html

HACK 8.10 [NOT] BETWEEN 句

BETWEEN 句もまた条件演算子の一種であり、大小比較可能な値に大して真とすべき範囲を指定するためのものである。従って WHERE 句などと共に使用する。この句で指定した境界値も真の範囲に含まれる。NOT を手前に付けると真偽が反転する。それゆえ、この句で指定した範囲はその境界値を含めて偽になる。

SQL 文の簡単な例を示す。

■SQL

```
-- 例1. 会員表"MEMBERS"に登録されている行のうち、
--      年齢列"age"の値が20代・30代の人をすべて出力
SELECT   *
FROM     "MEMBERS"
WHERE    "age" BETWEEN 20 AND 39;

-- 例2. 会員表"MEMBERS"に登録されている行のうち、入会年月日列"entdate"が
--      2010年4月1から2020年3月31日までの人をすべて出力
SELECT   *
FROM     "MEMBERS"
WHERE    "entdate" BETWEEN '2010-04-01' AND '2020-03-31';
```

UNIX の世界では、AWK コマンドの大小比較演算子を 2 回用いるのが妥当だろう。

例 2 に関しては年月日の桁が揃っていない（0 埋めされていない）可能性のあるデータであることを想定し、8.5.1 項の方針で数値化し、比較する例を示す。

■UNIX

```
# 例1. 会員表ファイル"MEMBERS.txt"に登録されている行のうち、
#      年齢列（5列目とする）の値が20代・30代の人をすべて出力
#     数字の1文字目が"5"である会員をすべて出力
awk '$5>=20 && $5<=39{print;}' "MEMBERS.txt"

# 例2. 会員表ファイル"MEMBERS.txt"に登録されている行のうち、
#      入会年月日列（11列目とする）が
#      2010年4月1から2020年3月31日までの人をすべて出力
cat "MEMBERS.txt"                                    |
awk '{s=$11;                                         #
      gsub(/[^0-9]/," ",s);                          #
      split(s,a);                                    #
      date=a[1]*10000+a[2]*100+a[3];                 #
      if(date>=20100401 && date<=20200331){print;}}'
```

HACK 8.11　[NOT] IN 句

IN 句または NOT IN 句は、“=”、“<”、“>” といった条件演算子の一種である。IN 句は、その直後で指定する値の集合にある要素のどれか 1 つ以上に一致すれば真になり、NOT IN 句は、その直後で指定する値の集合にある要素のどれにも一致しない場合のみ真になる。これらは条件演算子なので WHERE 句（HACK 8.5 参照）などと併用する。

値の集合としては即値を列挙することもできるし、副問い合わせによって表から得られる値を指定することもできる。

■SQL

```
-- 例1. 会員表"MEMBERS"に登録されている行のうち、
--      "id"列の値が試験用のもの('A0000','C0001','Z9999')を除いたものをすべて出力
SELECT   *
FROM     "MEMBERS"
WHERE    "id" NOT IN ('A0000','C0001','Z9999');

-- 例2. 会員表"MEMBERS"に登録されている行のうち、
--      "id"列の値が寄付者表"CONTRIBUTORS"に存在するもの
--        （ただし寄付年の列"year"の値が2021年のもの）のみ出力
SELECT   *
FROM     "MEMBERS"
WHERE    "id" IN (SELECT "id" FROM "CONTRIBUTORS" WHERE "year" = 2021);
```

UNIX の世界で IN 句や NOT IN 句に丁度いいものは、AWK コマンドの連想配列だ。そこでこれを使って次のように翻訳できる。

■UNIX

```
# 例1. 会員表ファイル"MEMBERS.txt"に登録されている行のうち、
#      ID列（1列目とする）の値が試験用のもの('A0000','C0001','Z9999')を
#      除いたものをすべて出力
#
cat "MEMBERS.txt"                 |
awk 'BEGIN      {id["A0000"]=1;   #
                 id["C0001"]=1;   #
                 id["Z9999"]=1;}  #
     !($1 in id){print;        }'

# 例2. 会員表ファイル"MEMBERS.txt"に登録されている行のうち、
#      ID列（1列目とする）の値が寄付者表ファイル"CONTRIBUTORS.txt"に存在するもの
#        （ただし寄付年の列（5列目とする）"year"の値が2021年のもの）のみ出力
awk '$5==2021{print $1;}' "CONTRIBUTORS.txt" > "$Tmp/contributors_2021"
cat "MEMBERS.txt"                                                    |
```

```
awk 'BEGIN   {while (getline n < "'"$Tmp/contributors_2021"'") {id[n]=1;}}  #
     $1 in id{print;                                                    }'
rm "$Tmp/contributors_2021"
```

ポイントは、IN 句や NOT IN 句で対象としていた値を連想配列のキーとして登録する点だ。値は何でもいい（この例では “1” とした）。そうすれば AWK の “in” 演算子を使って全く同じことができる。ただし、AWK に “not in” という演算子はないので NOT IN 句を翻訳する場合には “in” の評価式全体を “!( )” で囲う必要がある。

ところで例 2 では、一時ファイル（HACK 2.11 参照）を使っているが、もちろんこの程度の処理であれば 2 番目の AWK の BEGIN アクション内部で実装することで、一時ファイル不要にすることもできるのでそれでも構わない。ただ、どちらにしても処理の効率は大して変わらないと思う。そもそも例 2 のような処理が要求された場合には、私なら IN 句を使わずに INNER JOIN 句（HACK 9.2）を使うと思う。

HACK 8.12 ANY / SOME 句

ANY 句またはそれと同じ意味を持つ SOME 句は、IN 句とよく似た句ではあるが演算子未満の存在だ。IN 句は、作った値の集合の要素のどれかと「等しい」かどうかという評価まで含んでいるのに対し、ANY 句は単に集合を作って「この集合の要素うちのどれかと○○」と言うだけであり、○○に対応する条件演算子（“=”、“<”、“>” など）を ANY 句の手前に記す必要がある。（従って “= ANY” とすれば IN 句と同じ意味になる）

例えばこのような使い方ができる。

■SQL

```
-- 例. ジュニア会員表"JUNIORS"から、各人の賞の受賞回数"awards"列を抽出し、
--     正会員表"MEMBERS"と突き合わせ、
--     ジュニア会員の最多受賞回数未満の正会員の一覧を出力
SELECT   *
FROM     "MEMBERS"
WHERE    "awards" < ANY (SELECT "awards" FROM "JUNIORS");
```

つまり、ジュニア会員における最多受賞者の受賞回数が 10 回であるなら、正会員におて受賞回数 9 回以下の人が洗い出され、「お前ら頑張れ」という激励メッセージの宛名作成に利用されるという使い方だ。

これを UNIX の世界に翻訳する場合、IN 句の時のようにピタリと当てはまるコマンド等は思い当たらないものの、結局やりたい事は作った集合の中の最大値や最小値との比較であるのだから、そのように書けばいいだけだと思う。

■UNIX

```
# 例. ジュニア会員表ファイル"JUNIORS.txt"から、各人の賞の受賞回数列
#     （9列目とする）を抽出し、正会員表ファイル"MEMBERS.txt"と突き合わせ、
#     ジュニア会員の最多受賞回数未満の正会員の一覧を出力
jmax=$(cat "JUNIORS.txt"    |
       awk 'BEGIN{n= 0;   } #
            n>$9 {n=$9;   } #
            END  {print n;}')
```

```
awk -v jmax=$jmax '$9<jmax{print;}' "MEMBERS.txt"
```

HACK 8.13　ALL 句

ALL 句は ANY 句の仲間で、演算子未満の句と言える。両者の違いは ANY 句が、集合を作って「この集合の要素うちのどれかと○○」と言うのに対し、ALL 句は集合を作って「この集合の要素すべてと○○」と言うようなものである。（従って “!= ALL” とすれば NOT IN 句と同じ意味になる）

先程の正会員、ジュニア会員の例をもとにした応用例を示す。

■SQL

```
-- 例．ジュニア会員表"JUNIORS"から、各人の賞の受賞回数"awards"列を抽出し、
--     正会員表"MEMBERS"と突き合わせ、
--     ジュニア会員の最小受賞回数未満の正会員の一覧を出力
SELECT   *
FROM     "MEMBERS"
WHERE    "awards" < ALL (SELECT "awards" FROM "JUNIORS");
```

もしジュニア会員の全員が何回かの受賞経験があるにもかかわらず、正会員にその最少回数未満の受賞経験しかない人がいれば、そのような人が洗い出され、「お前らもっと頑張れ」という超激励メッセージの宛名作成に利用されるという使い方だ。

これを UNIX の世界に翻訳する場合も、ANY 句の場合と同様、結局やりたい事は作った集合の中の最大値や最小値との比較であるのだから、そのように書けばいいだけだと思う。

■UNIX

```
# 例．ジュニア会員表ファイル"JUNIORS.txt"から、各人の賞の受賞回数列
#      （9列目とする）を抽出し、正会員表ファイル"MEMBERS.txt"と突き合わせ、
#     ジュニア会員の最少受賞回数未満の正会員の一覧を出力
jmin=$(cat "JUNIORS.txt"              |
       awk 'BEGIN{n=(getline)?$9:0;}} #
            n<$9 {n=$9;             } #
            END  {print n;          }')
awk -v jmin=$jmin '$9<jmin{print;}' "MEMBERS.txt"
```

HACK 8.14　CASE 句

CASE 句は、CASE～WHEN～THEN～[WHEN～THEN～][ELSE～]END という構文の先頭に来るものである。広義の条件演算子と言え、条件によってこの構文が返す値が変わる。この構文を使う場所は主に SELECT コマンドの直後の出力列を指定する場所であり、例えば次のような使い方をする。

■SQL

```
-- 例．会員表"MEMBERS"の性別列"sex"の値（"M","F","X"）を「男」、「女」、「他」に
--     変換し、次の列構成で出力する。
--       1列目．"ID"（元表の"id"列そのまま）
```

```
--       2列目. "苗字"（元表の"Myoji"列そのまま）
--       3列目. "名前"（元表の"Namae"列そのまま）
--       4列目. "性別"（元表の"sex"列を漢字に変換）
--       5列目. "年齢"（元表の"age"列そのまま）
SELECT "id"                  AS "ID"  ,
       "Myoji"               AS "苗字",
       "Namae"               AS "名前",
       CASE "性別"
          WHEN 'M' THEN '男'
          WHEN 'F' THEN '女'
          ELSE          '他'
       END                   AS "性別",
       "age"                 AS "年齢"
FROM   "MEMBERS";
```

UNIX の世界で、CASE 構文に最も近いものは AWK コマンドの三項演算子だろう。従って、上記の SQL 文は次のように翻訳できる。

■UNIX

```
# 例. 会員表ファイル"MEMBERS"の性別列（6列目とする）の値（"M","F","X"）を
#      「男」、「女」、「他」に変換し、次の列構成で出力する。
#       1列目. 会員ID（元表の1列目列そのまま）
#       2列目. 苗字（元表の2列目列そのまま）
#       3列目. 名前（元表の3列目列そのまま）
#       4列目. 性別（元表の6列目を漢字に変換）
#       5列目. 苗字（元表の5列目列そのまま）
cat "MEMBERS"                      |
awk '{print $1                ,   #
            $2                ,   #
            $3                ,   #
            ($6=="M") ? "男" :    \
            ($6=="F") ? "女" :    \
                        "他"  ,   #
            $5                ;}'

# （三項演算子を用いない別解）
cat "MEMBERS"                             |
awk '{if        ($6=="M") {sex="男";    #
      } else if ($6=="F") {sex="女";    #
      } else              {sex="他";}   #
      print $1,$2,$3,sex,$5;         }'
```

このように、三項演算子は print 文に埋め込めるので SQL 文と近い感覚で使うことができる。しかし、

三項演算子を何重にもネストさせて使うことを好まない者もいると思うので[*3]、if 文を使う別解も記した。

HACK 8.15　LIMIT 句

LIMIT 句は SELECT コマンドの出力の直前に作用し、得られた結果のうち、先頭行から、あるいは指定行から指定された行数だけを出力するものである。出力されるデータをちょっとだけ見たい場合や、検索結果をページ分割する場合などに利用できる。

まずは SQL 文の例を示す。（ORDER BY 句については、HACK 9.1 参照）

■SQL

```
-- 例1. 会員表"MEMBERS"に登録されている行のうち、年齢列"age"が
--      20代・30代の人の行を選びだし、その結果のうち先頭の10名分だけを出力
SELECT   *
FROM     "MEMBERS"
WHERE    "age" BETWEEN 20 AND 39
LIMIT    10;

-- 例2. 会員表"MEMBERS"に登録されている行のうち、入会年月日列"entdate"が
--      2010年4月1日以降の人の行を選びだし、入会年月日列の古い順、次にID列"id"の
--      若い順にソートしたものを出力対象とし、今はこれを10人で1ページ分として
--      その場合の5ページ目（51行目からの10行）のみ出力
SELECT   *
FROM     "MEMBERS"
WHERE    "entdate" >= '2010-04-01'
ORDER BY "entdate" ASC,
         "id"      ASC
LIMIT    50,10;                    -- ※ LIMIT句の開始行番号は0から始まる
```

UNIX の世界で、先頭から指定行数を出力するコマンドといえば head である。また、指定行目から指定行数だけということであれば、tail コマンドで出力開始行番号を指定しつつ、その後 head コマンドで行数を指定すればよい。

■UNIX

```
# 例1. 会員表ファイル"MEMBERS.txt"に登録されている行のうち、
#      年齢列（5列目とする）が20代・30代の人の行を選びだし、
#      その結果のうち先頭の10名分だけを出力
cat "MEMBERS.txt"              |
awk '$5>=20 && $5<=39{print;}' |
head -n 10
```

[*3] `a?b:c?d:e` と書くと、AWK では `a?b:(c?d:e)` と解釈されるが、PHP では `(a?b:c)?d:e` と解釈されるなど、言語により演算子の優先順位が異なるから。

```
# 例2. 会員表ファイル"MEMBERS.txt"に登録されている行のうち、
#      入会年月日列（11行目とする）が2010年4月1日以降の人の行を選びだし、
#      入会年月日列の古い順、次にID列（1行目とする）の若い順にソートしたものを
#      出力対象とし、今はこれを10人で1ページ分として
#      その場合の5ページ目（51行目からの10行）のみ出力
#
#      ※ 入会年月日列は桁揃え済（YYYY/MM/DD形式）とする
#
cat "MEMBERS.txt"               |
awk '$11>="2010/04/01"{print;}' |
sort -b k 11,11 -k 1,1          |
tail -n +51                     |
head -n  10
```

第 9 章

SQL to UNIX マイグレーション (3) – ソート・結合の句

前章（選択に関する句）が長くなったので章を替え、この章ではソートの句（ORDER BY 句）と結合の句（〜JOIN）を UNIX の世界に翻訳（=UNIX コマンドで実装）する例を示していく。

この章でも POSIX 標準の UNIX コマンドのみならず、HACK 1.7 で紹介した “ShellShoccar” コマンドセット（Tukubai コマンド含む）が登場するので、その HACK を参照して準備しておくこと。

HACK 9.1　ORDER BY 句

ORDER BY 句は、選択の句で選び出された行の集合を、その集合が持つ列の値などの条件に応じてソートするものである。第 1 条件、第 2 条件、……と複数の条件を記述でき、手前の条件で順番が確定しなかった場合の追加条件が指定できる。またそれぞれの条件には昇順・降順が指定できる。“10” と “2” という 2 つの値を数値として見るか、文字列として見るかでソート結果は変わるが、SQL で管理されるデータベースにはデータ型があるのでどのように見なすかは自動的に決まる。

まず SQL 文の例を示す。

■SQL

```
-- 例．会員表"MEMBERS"に登録されている行を、次の条件でソートして出力
--       第1条件．年齢列"age"の高い順
--       第2条件．苗字のふりがな列"MyojiYomi" の50音順
--       第3条件．名前のふりがな列"NamaeYomi" の50音順
--     ただし、出力する列は1列から順に、次の通りとする（ふりがなは出力しない）
--       1列目．会員ID列"id"
--       2列目．年齢　列"age"
--       3列目．苗字（漢字）列"Myoji"
--       4列目．名前（漢字）列"Namae"
SELECT   "id"   ,
         "age"  ,
         "Myoji",
```

```
        "Namae"
FROM    "MEMBERS"
ORDER BY "age"       DESC,
        "MyojiYomi" ASC ,
        "NamaeYomi" ASC ;
```

UNIX の世界に翻訳する際、ソートといえば sort コマンドである。そこで sort コマンドを使ってこれを書くと次のようになる。

■UNIX

```
# 例．会員表ファイル"MEMBERS.txt"に登録されている行を、次の条件でソートして出力
#       第1条件．年齢列（5列目）とするの高い順
#       第2条件．苗字のふりがな列（7列目とする） の50音順
#       第3条件．名前のふりがな列（8列目とする） の50音順
#     ただし、出力する列は1列から順に、次の通りとする（ふりがなは出力しない）
#       1列目．会員ID列（会員表ファイルの1列目）
#       2列目．年齢　列（会員表ファイルの5列目）
#       3列目．苗字（漢字）（会員表ファイルの2列目）
#       4列目．名前（漢字）（会員表ファイルの3列目）
#
#     ※ コマンドの間にあるコメント行（"1:ID 2:年齢"など）は、
#        その位置を流れるデータの列構成を表している
#
cat "MEMBERS.txt"                                |
self 1 5 2 3 7 8                                 |
# 1:ID 2:年齢 3:苗字 4:名前 5:苗字読み 6:名前読み #
sort -b -k 2nr,2 -k 5,5 -k 6,6                   |
self 1/4
```

この例では self コマンド（列の選択）を 2 段階で行っている点に注目しなければならない。

処理対象のデータを減らすため、なるべく早い段階で表示対象の列のみに絞り込みたいところではあるが、ふりがなの列は sort コマンドで必要になるので、最初の段階で除去するわけにはいかない。sort が済んでから不要になるので、sort コマンドの後でもう一度 self コマンドを使っている。この点は SQL 文では考える必要がなかったことなので注意すること。

HACK 9.2 INNER JOIN 句

INNER JOIN 句は、2 つの表（1:左表、2:右表）を内部結合するための句である。内部結合なので、両方の表の行が同じ条件を同時に満たせるもの同士だけが結合されて残り、そうでない行は結合後の行には残らない。

SQL 文の例を示す。

■SQL

```
-- 例．会員IDをはじめ、詳細な会員情報が格納されている表"MEMBERS"と
```

```
--      2020年の月例イベントへの各会員の参加状況を格納した表"ATTEND2020"がある。
--      "ATTEND2020"は1列目が会員ID列"id"で、2〜13の各列の名前は"1"〜"12"であり
--      各月のイベントの出欠情報（0なら欠席、1なら出席）が入っている。
--      （ただし、"ATTEND2020"には年間1度も出席しなかった人の行は無い）
--      この"MEMBERS"と"ATTEND2020"を、会員ID列"id"の一致を条件に内部結合し、
--      次の列順で出力する
--       1列目．会員ID列"id"
--       2列目．苗字（漢字）列"Myoji"
--       3列目．名前（漢字）列"Namae"
--       4列目．年間の出席回数（出欠情報列の合計値）
SELECT   MEM."id"                                                    AS "id",
         MEM."Myoji"                                                 AS "Myoji",
         MEM."Namae"                                                 AS "Namae",
         ATT."1" + ATT."2" + ATT."3" + ATT."4" + ATT."5" + ATT."6" +
         ATT."7" + ATT."8" + ATT."9" + ATT."10"+ ATT."11"+ ATT."12" AS "times"
FROM     "MEMBERS"    AS MEM
           INNER JOIN
         "ATTEND2020" AS ATT
           ON MEM."id" = ATT."id";
```

UNIX の世界では、結合（○○ JOIN）といえば join コマンドの出番である。join コマンドを使って上記の SQL 文を翻訳すると次のようになる。

■UNIX

```
# 例．会員IDをはじめ、詳細な会員情報が格納されている表ファイル"MEMBERS.txt"
#      （1列目にある会員ID列でソート済）と、2020年の月例イベントへの各会員の
#      参加状況を格納した表ファイル"ATTEND2020.txt"がある。
#      "ATTEND2020.txt"は1列目が会員ID列で、2〜13の各列は各月のイベントの
#      出欠情報（0なら欠席、1なら出席）が入っている。
#      （ただし、年間1度も出席しなかった人の行は無い）
#      この"MEMBERS.txt"と"ATTEND2020.txt"を、会員ID列（どちらも1列目とする）の
#      一致を条件に内部結合し、次の列順で出力
#       1列目．会員ID列（1列目）
#       2列目．苗字（漢字）列（"MEMBERS.txt"の2列目）
#       3列目．名前（漢字）列（"MEMBERS.txt"の3列目）
#       4列目．年間の出席回数（"ATTEND2020.txt"の2〜13列目の合計値）
#
#      ※ コマンドの間にあるコメント行（"1:ID 2:苗字"など）は、
#         その位置を流れるデータの列構成を表している
#
cat "ATTEND2020.txt"                                                 |
sort -bk 1,1                                                         |
join -1 1 -2 1                                                       \
```

```
    -o 1.1,1.2,1.3,2.2,2.3,2.4,2.5,2.6,2.7,2.8,2.9,2.10,2.11,2.12,2.13 \
    "MEMBERS.txt" -                                                    |
# 1:ID 2:苗字 3:名前 4-15:各月の出欠情報                               #
awk '{times=0;                                                       #
     for (i=1;i<=12;i++) {times+=$(i+3);}                            #
     print $1,$2,$3,times;              }'
```

最初に join コマンドの記述内容について説明する。

コマンドの直後に “-1” と “-2” というオプションがあるが、これらの数字は表に振られた番号である。1 が左表、2 が右表を意味する。そしてそれぞれのオプション引数の番号は、それぞれの表内の何列目の一致を結合条件にするかを指定するためのものである。今はどちらも 1 列目にある会員 ID なので、両方とも “1” である。

次に “-o” オプションは、結合によってできた行の新たな列構成を指定するためのものである。データの由来する表番号と列番号をドット “.” で繋いだものを、カンマ “,” 区切りで書き連ねる。今の場合は、会員 ID 列（”MEMBERS.txt”の 1 列目）、苗字列（同 2 列目）、名前列（同 2 列目）、そして、出欠情報列（”ATTEND2020.txt”の 2〜13 列目）を並べている。

最後は、結合対象となる表ファイルを左表、右表の順に書く。この例では右表は標準入力から到来するデータとするので “-” としてある。

こうして結合された行データを生成したら次の AWK コマンドに与え、12 個ある出欠情報列の値を合計し、会員 ID と苗字、名前と共に出力する。その AWK の部分は、Tukubai コマンドの 1 つである “ysum”[*1]を使ってもっと簡単に書くこともできる。

■UNIX – “ysum” コマンドによる例

```
# （Tukubaiコマンドの1つ"ysum"を使った別解）
cat "ATTEND2020.txt"                                                 |
sort -bk 1,1                                                         |
join -1 1 -2 1                                                       \
     -o 1.1,1.2,1.3,2.2,2.3,2.4,2.5,2.6,2.7,2.8,2.9,2.10,2.11,2.12,2.13 \
     "MEMBERS.txt" -                                                 |
# 1:ID 2:苗字 3:名前 4-15:各月の出欠情報                             #
yum num=3                                                            |
# 1:ID 2:苗字 3:名前 4-15:各月の出欠情報 16:出席回数(4-15列目の合計値) |
self 1 2 3 16
```

“ysum” とは「横サム」、つまり横に並んだ列の合計という意味であり、1 列目から “num=” オプションで指定した列番号までをキー列とみなし、その次の列から最終列までを合計したものを、最終列の後ろに置く。今回の例では各月の出欠情報は表示する必要がないので、ysum コマンドの後で、self コマンドによって消している。

[*1] 詳細は次の URL を参照。
https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_ysum

9.2.1　結合条件が複数列に及ぶ場合

join コマンドの結合条件の指定方法は “-1” と “-2” オプションによるものであり、これでは複数列の一致を結合条件とすることができない。

例えば、ある会では会員 ID が存在せず、苗字と名前と入会年月日の 3 つの組み合わせて一意性を確保していたとすると、何かの表と結合する時には、苗字・名前・入会年月日の 3 つとも一致することを確認しなければならない。しかし、SQL 文なら特に難しいことはない。

■SQL

```
-- 例．次のような構成で会員情報が格納されている表"MEMBERS"と
--       1列目．苗字（漢字）列"Myoji"
--       2列目．名前（漢字）列"Namae"
--         :
--       10列目．入会年月日列"entdate"
--      2020年の月例イベントへの各会員の参加状況を格納した表"ATTEND2020"がある。
--       1列目．苗字（漢字）列"Myoji"
--       2列目．名前（漢字）列"Namae"
--       3列目．入会年月日列"entdate"
--       4-15列目．列名"1"～"12"で1～12月各月の出欠情報（0なら欠席、1なら出席）
--       （ただし、"ATTEND2020"には年間1度も出席しなかった人の行は無い）
--      この"MEMBERS"と"ATTEND2020"を、苗字・名前・入会年月日列すべての一致で結合し
--      次の列順で出力
--       1列目．苗字（漢字）列"Myoji"
--       2列目．名前（漢字）列"Namae"
--       3列目．入会年月日列"entdate"
--       4列目．年間の出席回数（出欠情報列の合計値）
SELECT   MEM."Myoji"                                                    AS "Myoji"   ,
         MEM."Namae"                                                    AS "Namae"   ,
         MEM."entdate"                                                  AS "entdate",
         ATT."1" + ATT."2" + ATT."3" + ATT."4" + ATT."5" + ATT."6" +
         ATT."7" + ATT."8" + ATT."9" + ATT."10"+ ATT."11"+ ATT."12" AS "times"
FROM     "MEMBERS"    AS MEM
           INNER JOIN
         "ATTEND2020" AS ATT
           ON MEM."Myoji"   = ATT."Myoji"   AND
           ON MEM."Namae"   = ATT."Namae"   AND
           ON MEM."entdate" = ATT."entdate"    ;
```

UNIX の世界に翻訳するには、join コマンドの使い方を工夫する。結合条件に複数列を指定できないのなら、対象となるすべての列の文字列を結合して 1 列にまとめた新たな列を一時的に作り、それを join コマンドに与えればよい。

というわけでこの方針で翻訳したものが次のコードである。

■UNIX

```
# 例．次のような構成で会員情報が格納されている表ファイル"MEMBERS.txt"と
#       1列目．苗字（漢字）列
#       2列目．名前（漢字）列
#         :
#       10列目．入会年月日列
#       （苗字→名前→入会年月日の順にソート済とする）
#     2020年の月例イベントへの各会員の参加状況を格納した表ファイル
#     "ATTEND2020.txt"がある。
#       1列目．苗字（漢字）列
#       2列目．名前（漢字）列
#       3列目．入会年月日列
#       4-15列目．1〜12月各月の出欠情報（0なら欠席、1なら出席）
#       （苗字→名前→入会年月日の順にソート済とする）
#       （ただし、"ATTEND2020"には年間1度も出席しなかった人の行は無い）
#     この"MEMBERS.txt"と"ATTEND2020.txt"を、苗字列、名前列、入会年月日列
#     すべての一致を条件に内部結合し、次の列順で出力
#       1列目．苗字（漢字）列
#       2列目．名前（漢字）列
#       3列目．入会年月日列
#       4列目．年間の出席回数（出欠情報列の合計値）
#
#     ※ コマンドの間にあるコメント行（"1:苗字+名前+入会年月日 2:苗字"など）は、
#         その位置を流れるデータの列構成を表している
#
cat "MEMBERS.txt"                                    |
awk '{print $1 "\034" $2 "\034" $10, $1, $2, $10;}' > "$Tmp/members"
# 1:苗字+名前+入会年月日 2:苗字 3:名前 4:入会年月日
#
cat "ATTEND2020.txt"                                                    |
awk '{print $1 "\034" $2 "\034" $10, $0;}'                              |
# 1:名前+苗字+入会年月日 2:苗字 3:名前 4:入会年月日 5-16:出欠情報          #
join -1 1 -2 1                                                          \
     -o 1.2,1.3,1.4,2.5,2.6,2.7,2.8,2.9,2.10,2.11,2.12,2.13,2.14,2.15,2.16 \
     "$Tmp/members" -                                                   |
# 1:苗字 2:名前 3:入会年月日 4-15:各月の出欠情報                          #
yum num=3                                                               |
# 1:ID 2:苗字 3:名前 4-15:各月の出欠情報 16:出席回数(4-15列目の合計値)     |
self 1 2 3 16
#
rm "$Tmp/members"
```

ポイントは 2 つある。

1 つは複数列の文字列結合を行うと、どうしても一時ファイルを使わなければならなくなること。もう 1 つは、複数列を結合する際には結合される列にはどこにも含まれていない列区切り文字を用意し、それを間に挿みながら結合することである。

その際に用いるお勧めの文字は、制御文字の 1 つである Field Separator 文字（0x1C、8 進数だと 034）である。このような制御文字は普通のテキストデータにはおよそ登場し得ない文字であるので安心して使えるうえ、制御文字領域の文字はテキストデータに登場する文字よりも文字コードの順位が若いので、**文字列結合前の各列がソート済であるなら結合後に再ソート処理を行う必要が無い**からである。

なお、次項で事前ソートができない場合の対応策を示すが、そちらで紹介する方法を使う場合には始めから複数列に渡る条件指定が可能なので、ここで紹介したテクニックは使わなくてよい。

9.2.2　行数が膨大であるなどの事情で、事前ソートが困難な場合

例えば、会員のみがログインできる Web ページがあって、ログインからログアウトまでその会員が行った操作内容が時系列順に記録されているアクセスログ表があったとする。列構成は 1 列目が日時、2 列目が会員 ID、3 列目が操作内容であり、ページ移動から何から何まで記録されているために行数は膨大である。

会員 ID だけでは、誰の操作なのかがわかりにくいので会員表を（内部）結合して会員氏名を添えたいものの、アクセスログ表の行数は膨大だし、結合した後の表は元の時系列順で欲しいという要求仕様だったとすると、join のためとはいえどアクセスログ表をソートするのは得策ではない。

まずは SQL 文の例を示す。SQL 文では JOIN 句を使う際に事前ソートをする必要がない[*2]のでまったく問題なく書けてしまう。

■SQL

```
-- 例．アクセスログ表"ACCESSLOG"
--       1列目．日時列"time"
--       2列目．会員ID列"id"
--       3列目．操作内容"operation"
--      に対し、会員表"MEMBERS"
--       1列目．会員ID列"id"
--       2列目．苗字（漢字）列"Myoji"
--       3列目．名前（漢字）列"Namae"
--                  :
--      の会員ID列を内部結合し、次の列構成で出力
--       1列目．日時列"time"
--       2列目．会員ID列"id"
--       3列目．苗字（漢字）列"Myoji"
--       4列目．名前（漢字）列"Namae"
--       5列目．操作内容"description"
SELECT   ACC."time"         AS "time"        ,
         MEM."id"           AS "id"          ,
```

*2 内部的にはソートを行っている（いない場合もある）が、ユーザーには見えないだけである。

```
        MEM."Myoji"       AS "Myoji"      ,
        MEM."Namae"       AS "Namae"      ,
        ACC."description" AS "description"
FROM    "MEMBERS"    AS MEM
          INNER JOIN
        "ACCESSLOG"  AS ACC
          ON MEM."id" = ACC."id"
ORDER BY ACC."time";
```

では UNIX の世界ではどうすべきかというと、AWK の連想配列を使うのがよいと思う。

この例の場合は、会員表の ID 列をキーにして、その値としては苗字と名前を半角空白で繋いだ文字列を代入すればよい。その登録処理を AWK の BEGIN セクションの中で行い、通常のセクションでは到来した会員 ID 列の値がキーとして存在するものである場合にのみ、先程の連想配列の値を列の途中に挿入しながら出力する。

このやり方はソートを必要としないというメリットがある一方、会員表ファイルのすべての行をメモリ上に読み込んでおく必要があるので、会員表が多い場合にはメモリを大量消費するというデメリットを伴う。

■UNIX

```
# 例．アクセスログ表ファイル"ACCESSLOG.txt"
#      （各行は時間順に並んでいるとする）
#      1列目．日時列
#      2列目．会員ID列
#      3列目．操作内容
#     に対し、会員表ファイル"MEMBERS.txt"
#      1列目．会員ID列
#      2列目．苗字（漢字）列
#      3列目．名前（漢字）列
#               :
#     の会員ID列を内部結合し、次の列構成で出力
#      1列目．日時列
#      2列目．会員ID列
#      3列目．苗字（漢字）列
#      4列目．名前（漢字）列
#      5列目．操作内容
cat "ACCESSLOG.txt"                                                    |
awk 'BEGIN     {while(getline < "ACCESSLOG.txt") {name[$1] = $2 " " $3;}  #
     $1 in name{print $1,$2,name[$1],$3;                                  }'
```

なお、ソート回避のために AWK を使おうとしていて、かつ結合条件も複数列に及ぶという場合（9.2.1 項と 9.2.2 項の複合ケース）、AWK の連想配列を多次元で使えばよい。例えば、上記の例で結合条件が 1 列と 2 列目であった場合次のような置き換えをすればいい。

- `name[$1]`（2 箇所） → `name[$1 "\034" $2]`
  - 角括弧の中に限りカンマ区切りで書ける（`name[$1,$2]` と書いても同じ）
- `n$1 in name` → `($1 "\034" $2) in name`

このようにして AWK コマンドを使えば対応可能ではあるが、少々ごちゃごちゃしてしまう。Tukubai コマンドセットには、このような場面を想定した cjoin というコマンドのシリーズがある。具体的には “cjoin0”、“cjoin1”、“cjoin2” であり、このうち内部結合に用いるのは cjoin1[*3]である。HACK 1.7 でインストールしたであろう ShellShoccar コマンドセットにも cjoin シリーズが移植してあり、この移植版もやはり AWK の連想配列を使って実装している。

ということで、cjoin1 を使った翻訳例を示す。

■UNIX – cjoin1 コマンド使用

```
# 例．アクセスログ表ファイル"ACCESSLOG.txt"
#       （各行は時間順に並んでいるとする）
#       1列目．日時列
#       2列目．会員ID列
#       3列目．操作内容
#     に対し、会員表ファイル"MEMBERS.txt"
#       1列目．会員ID列
#       2列目．苗字（漢字）列
#       3列目．名前（漢字）列
#                 :
#     の会員ID列を内部結合し、次の列構成で出力
#       1列目．日時列
#       2列目．会員ID列
#       3列目．苗字（漢字）列
#       4列目．名前（漢字）列
#       5列目．操作内容
#
#     ※ コマンドの間にあるコメント行（"1:日時 2:会員ID"など）は、
#        その位置を流れるデータの列構成を表している
#     ※ 列番号の"NF"とは最終列番号を意味している
#
cjoin1 key=2 "MEMBERS.txt" "ACCESSLOG.txt"                                    |
# 1:日時 2:会員ID 3:苗字 4:名前 5〜NF-1:会員表ファイルの4列目以降 NF:操作内容 #
self 1 2 3 4 NF
# 1:日時 2:会員ID 3:苗字 4:名前 5:操作内容
```

cjoin シリーズの “key=” オプションには **2 番目の表ファイルの**結合条件列番号を記す。なお、結合条件が複数列に渡る場合は “m/n” のようにして範囲指定できるので 9.2.1 項で紹介したようなテクニックはこのコマンドに対しては不要である。なお、1 番目の表ファイルの結合条件列は 1 列目固定（複数列ある場合は 1 列目から “key=” オプションで指定した個数）である。そして結合後の列構成は、2 番目の表ファイルの “key=” オプションで指定した列番号の部分に、結合された 1 番目の表ファイルの 1 行分が丸ごと挿入されたものになる。

[*3] 詳細は次の URL を参照。
https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_cjon1

このように結合条件列や結合後の列構成が固定されているので、join コマンドと違って簡潔に記述できる。その代わり、cjoin シリーズに与えるデータや出力されるデータは適宜列の入れ替えや選択が必要なので、この例でも直後に self コマンドを置いて指定された列を取り出している。

HACK 9.3 LEFT [OUTER] JOIN / RIGHT [OUTER] JOIN 句

LEFT [OUTER] JOIN 句も RIGHT [OUTER] JOIN 句も、2 つの表（1:左表、2:右表）を外部結合するための句である。違いは左外部結合か右外部結合かであり、左右の表で条件を満たさない行があった場合に、左外部結合では左表にある行のみが結合相手が無いまま結合後の表に出力され、右外部結合では右表にある行のみが結合相手が無いまま結合後の表に出力される。

ここでは 9.2.2 項で示した、アクセスログ表に会員名を付け足す例で考える。もし会員表には存在しない会員 ID の行がなぜかアクセスログ表にあったとして、会員表には無いからという理由で消えてしまっては困ることの方が多いだろう。もしかしたら不正アクセスの証拠なのかもしれないのだから。

というわけで、外部結合してアクセスログ表にある行は消さない SQL 文の例を示す。

■SQL

```
-- 例．アクセスログ表"ACCESSLOG"（左表とする）
--      1列目．日時列"time"
--      2列目．会員ID列"id"
--      3列目．操作内容"operation"
--     に対し、会員表"MEMBERS"（右表とする）
--      1列目．会員ID列"id"
--      2列目．苗字（漢字）列"Myoji"
--      3列目．名前（漢字）列"Namae"
--                :
--     の会員ID列を左外部結合し、次の列構成で出力
--      1列目．日時列"time"
--      2列目．会員ID列"id"
--      3列目．苗字（漢字）列"Myoji"
--      4列目．名前（漢字）列"Namae"
--      5列目．操作内容"description"
SELECT   ACC."time"        AS "time"       ,
         MEM."id"          AS "id"         ,
         MEM."Myoji"       AS "Myoji"      ,
         MEM."Namae"       AS "Namae"      ,
         ACC."description" AS "description"
FROM     "MEMBERS"    AS MEM
           LEFT OUTER JOIN
         "ACCESSLOG"  AS ACC
           ON MEM."id" = ACC."id"
ORDER BY ACC."time";
```

これが UNIX の世界へ翻訳されるとどうなるかだが、その前にまず NULL 値の扱いを確認しなければ

ならない。SQL 文で外部結合をして結合相手の無い行が生じると、本来結合相手の表から代入されるはずの列には NULL 値が入る。しかし HACK 2.4 で解説したように、UNIX の世界のテキストデータでは NULL 値を直接表現することができない。したがって外部結合をする時には、その HACK に記した方針に基づいて、NULL 値と見なすことにする文字を必ず決めなければならない。

さらに UNIX の世界への翻訳を考える時、事前ソートが可能か不可能かを考えなければならないが、まずここでは可能な場合の例を考える。可能であるなら join コマンドが使える。

■UNIX

```
# 例．アクセスログ表ファイル"ACCESSLOG.txt"
#     （左表かつ会員ID順にソートしても問題ないものとする）
#       1列目．日時列
#       2列目．会員ID列
#       3列目．操作内容
#     に対し、会員表ファイル"MEMBERS.txt"
#     （右表かつ会員ID順にソート済とする）
#       1列目．会員ID列
#       2列目．苗字（漢字）列
#       3列目．名前（漢字）列
#                 :
#     の会員ID列を左外部結合（ただしNULL列は"*"で表現）し、次の列構成で出力
#       1列目．日時列
#       2列目．会員ID列
#       3列目．苗字（漢字）列
#       4列目．名前（漢字）列
#       5列目．操作内容
#
#     ※ コマンドの間にあるコメント行（"1:日時 2:会員ID"など）は、
#        その位置を流れるデータの列構成を表している
#
cat "ACCESSLOG.txt"                                                  |
# 1:日時 2:会員ID 3:操作内容                                         #
sort -bk 2,2                                                         |
    join -1 1 -2 1 -a 1 -e '*' -o 2.1,1.1,1.2,1.3,2.3 - "MEMBERS.txt" |
# 1:日時 2:会員ID 3:苗字 4:名前 5:操作内容                           #
sort -bk 1,1
```

このように、事前ソートが可能なら右または左の外部結合にも join コマンドが使える。ただ、内部結合の時には使わなかった 2 つの新たなオプションが必要になる。

1 つ目は “-a” オプションだ。直後には 1 か 2 どちらかの数字が続き、“-a 1” なら左外部結合（1 番目のファイルにある結合できなかった行を出力）であり、“-a 2” なら右外部結合（2 番目のファイルにある結合できなかった行を出力）である。

2 つ目は “-e” オプションで、直後には結合できずに NULL 値が入ることになった列に与える文字（文字列でもよい）を指定するためのものである。この例では NULL 値と見なす値を “*” としている。ちなみ

にこのオプションは省略可能だが、省略すると空文字になってしまう。列区切りの半角空白は 1 文字とするという取り決めをしているならば、半角空白文字が何個連続しているかを数えることで NULL 値が置かれている箇所を認識できなくはないが、AWK をはじめとした UNIX コマンドの多くはそういうルールにはなっていないので事故のもと。省略してはならない。

そして、join コマンドの後にはソートを忘れずに行う。前処理として会員 ID 順にソートして、元々のアクセスログ表の順番を崩したからだ。

9.3.1 結合条件が複数列に及ぶ場合

9.3.2 行数が膨大であるなどの事情で、事前ソートが困難な場合

次に、これら 2 つの問題についてはまとめて取り扱う。

基本的にはどちらも内部結合の時と同じ考え方で対応できるので、9.2.1 項と 9.2.2 項を読んで応用すればいい。join コマンドを使う場合には、“-a” と “-e” オプションを追加するという点が違うだけだし、ソート回避のために AWK コマンドを使うならの連想配列を使えばいいだけだ。

しかし、9.2.2 項での紹介した Tukubai コマンドセットの cjoin シリーズを使えば、これらの問題は同時に解決でき、しかも簡単な記述で済むようになる。cjoin シリーズで外部結合用のものは cjoin2 コマンド[*4]である。

上記のアクセスログ外部結合の例を cjoin2 コマンドで実装すると次のようになる。

■UNIX

```
# 例．アクセスログ表ファイル"ACCESSLOG.txt"
#     （左表かつ会員ID順にソートしても問題ないものとする）
#       1列目．日時列
#       2列目．会員ID列
#       3列目．操作内容
#     に対し、会員表ファイル"MEMBERS.txt"
#     （右表かつ会員ID順にソート済とする）
#       1列目．会員ID列
#       2列目．苗字（漢字）列
#       3列目．名前（漢字）列
#                 :
#     の会員ID列を左外部結合（ただしNULL列は"*"で表現）し、次の列構成で出力
#       1列目．日時列
#       2列目．会員ID列
#       3列目．苗字（漢字）列
#       4列目．名前（漢字）列
#       5列目．操作内容
#
```

[*4] 詳細は次の URL を参照。
https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_cjon2

```
#     ※ コマンドの間にあるコメント行（"1:日時 2:会員ID"など）は、
#        その位置を流れるデータの列構成を表している
#     ※ 列番号の"NF"とは最終列番号を意味している
#
cjoin2 key=2 +'*' "MEMBERS.txt" "ACCESSLOG.txt"                              |
# 1:日時 2:会員ID 3:苗字 4:名前 5〜NF-1:会員表ファイルの4列目以降 NF:操作内容 #
self 1 2 3 4 NF
# 1:日時 2:会員ID 3:苗字 4:名前 5:操作内容
```

cjoin2 コマンドで NULL 値文字を指定するには “+” というオプションの直後（空白を入れずに）にその文字を記す。そのため、この例では “+*” としたいところだが、アスタリスクはワイルドカードと解釈される恐れがあるので “+'*'” と記している。

次に、cjoin2（を含む cjoin シリーズ全体）では、どちらの表データを 1 番目（左表）、2 番目（右表）にするかに作法があるので知っておいてもらいたい。cjoin シリーズでは一般的に、**マスターデータを左表、トランザクションデータを右表**にする。従って、この例では会員表ファイルを左表（1 番目の表）としている。

条件列番号指定のための “key=” オプションが右表（つまりトランザクションデータ表）用のものであって、左表（マスターデータ表）は 1 列目からに固定されているのは、恐らく次のような理由であろう。マスター表はトランザクション表に対して都合のいいように事前にソートや列入れ替えを行っておけばいいが、トランザクション表はそれらの加工が（計算コスト的にも）面倒だろうということで、このような仕様になっているものと思われる。

HACK 9.4　FULL [OUTER] JOIN 句

FULL [OUTER] JOIN 句は、2 つの表（1:左表、2:右表）を完全外部結合するための句である。右または左外部結合では、左右の表で結合条件を満たさない行があった場合に、右外部結合なら右表、左外部結合なら左表の行のみを結合後の表に含めていたが、完全外部結合ではどちらの表の行も含ませる。

引き続き、9.2.2 項で示した、会員表とアクセスログ表を例に考えることにしよう。

今、アクセスログ表をもとに、各会員の操作回数を集計したいとする（ID のみならず名前を含めて一覧表示）。基本的にはアクセスログ表の会員 ID 列で集計を取ればいいが、それでは操作回数 0 だった会員の行は作られない。ならば会員表側の行をすべて残す形の外部結合を行えばよいように思うかもしれないが、アクセスログ表には不正アクセスなどによって会員表には存在しない ID の行が記録されている可能性もあり、そういった行もちゃんと集計したい。……と、このようなケースで使えるだろう。

それではまず、この要求に基づいた SQL 文の例を示す。（文中に出てくる SUM 関数や IFNULL 関数は、それぞれ HACK 10.45、HACK 10.19 を参照）

■SQL

```
-- 例．アクセスログ表"ACCESSLOG"
--       1列目．日時列"time"
--       2列目．会員ID列"id"
--       3列目．操作内容"operation"
--      と、会員表"MEMBERS"
--       1列目．会員ID列"id"
--       2列目．苗字（漢字）列"Myoji"
```

```
--      3列目．名前（漢字）列"Namae"
--                :
--      の会員ID列を完全外部結合し、同一会員ID列値の出現回数を集計
--      ただし、会員表由来でアクセスログ表になかった行の出現回数は0とする
--      最終的に出力する列の構成は次の通りとし
--      1列目．会員ID列"id"
--      2列目．操作回数列"Myoji"
--      3列目．操作回数列"Namae"
--      4列目．操作回数列"times"
--      回数の多い順→会員ID順でソートして出力
SELECT   AC2."id"         AS "id"   ,
         AC2."Myoji"      AS "Myoji",
         AC2."Namae"      AS "Namae",
         SUM(AC2."times") AS "times"  -- SUM()は集約対象列の合計値を得る関数
FROM     (SELECT IFNULL(AC1."id",MEM."id") AS "id"   , -- IFNULL()は,NULL値だったら
                 IFNULL(MEM."Myoji",'*')   AS "Myoji", -- 代替値を返す関数
                 IFNULL(MEM."Namae",'*')   AS "Namae", --  （同等の関数は製品により
                 IFNULL(AC1."times", 0 )   AS "times"  --   バラバラな仕様）
          FROM   (SELECT "id" AS "id"      ,
                         1    AS "times"
                  FROM   "ACCESSLOG"        ) AS AC1
                   FULL OUTER JOIN
                 (SELECT "id" AS "id"      ,
                         "Myoji" AS "Myoji",
                         "Namae" AS "Namae"
                  FROM   "MEMBERS"          ) AS MEM
                   ON ON AC1."id" = MEM."id"        ) AS AC2
GROUP BY AC2."id"   ,
         AC2."Myoji",
         AC2."Namae"
ORDER BY "time" DESC,
         "id"   ASC;
```

副問い合わせが二重にあって複雑であるが、やっていることは次の通りである。

二重の副問い合わせになっている一番インデントされている SELECT 文では、アクセスログ表からは会員 ID 列と、回数として 1 という数字を即値で、という以上 2 つの列を取り出している。この回数列”times”を会員 ID 毎に足し合わせて各人の操作回数を得ようという方針だ。そして、同じ階層のもう 1 つ SELECT 文からは、会員の苗字・名前を取り出してている。これらを完全外部結合してアクセスログ表由来の会員 ID に名前を（付けられれば）付けている。その外側の SELECT 文では、NULL 値の解決を図っている。アクセスログ表由来で、会員表に氏名の情報が無かった列の苗字と名前は “*” とし、逆に会員表由来で一度も操作実績が無い行ゆえに”times”列の値が NULL になっているものには 0 を与えている。そして、一番外側の SELECT 文で、会員 ID（および苗字と名前）ごとに集計をして操作回数の合計

を得ている。最後は、その操作回数と会員 ID でソートをし、出力とする。

では、これを UNIX の世界に翻訳するとどう書けるかというと、これまでの例と同様に join コマンドまたは AWK コマンドで書ける。

まず、アクセスログ表ファイルが事前ソート可能であり、join コマンドが使える例から見ていく。

■UNIX – 事前ソート可能で join コマンドを使う場合

```
# 例．アクセスログ表ファイル"ACCESSLOG.txt"（会員ID列でソート可能な規模とする）
#       1列目．日時列
#       2列目．会員ID列
#       3列目．操作内容
#     と、会員表ファイル"MEMBERS.txt"（会員ID列でソート済とする）
#       1列目．会員ID列
#       2列目．苗字（漢字）列
#       3列目．名前（漢字）列
#                 :
#     の会員ID列を完全外部結合し、同一会員ID列値の出現回数を集計
#     ただし、会員表由来でアクセスログ表になかった行の出現回数は0とする
#     最終的に出力する列の構成は次の通りとし
#       1列目．会員ID列
#       2列目．操作回数列
#       3列目．操作回数列
#       4列目．操作回数列
#     回数の多い順→会員ID順でソートして出力
#
#     ※ コマンドの間にあるコメント行（"1:日時 2:会員ID"など）は、
#        その位置を流れるデータの列構成を表している
#     ※ 列番号の"NF"とは最終列番号を意味している
#
cat "ACCESSLOG.txt"                                                  |
awk '{print $2,1;}'                                                  |
# 1:会員ID 2:回数(=1)                                                #
sort -bk 2,2                                                         #
join -1 1 -2 1 -a 1 -a 2 -e '*' -o 1.1,2.1,1.2,1.3 "MEMBERS.txt" - |
# 1:会員ID(会員表由来) 2:会員ID(ログ由来) 3:苗字 4:名前 5:回数(=1)  #
awk '{id =($1!='*')?$1:$2; # NULLではない列の会員IDを採用           #
      num=($5!='*')?$5: 0; # 回数がNULLなら0を代入                  #
      print id, $3, $4, num;                                  }'   |
# 1:会員ID 2:苗字 3:名前 4:回数(0か1)                               #
sm2 1 3 4 4                                                          |
# 1:会員ID 2:苗字 3:名前 4:回数(集計済)                            #
sort -b -k 4nr,4 -k 1,1
```

SQL 文での方針が把握できていれば UNIX 版を理解するのも容易なはずだ。

最初の AWK では、アクセスログ表ファイルのうち会員 ID 列だけと取り出しつつ、回数として 1 を与えている。このようにして、処理に不要な行は早い段階で取り除いておくことが高速化に繋がる。

次にそれを会員 ID でソート、そして join コマンドに右表として与えている。ここの join コマンドで注目すべきは、“-a” オプションを 2 回使っている点だ。“-a 1 -a 2” とすることで左右どちらの結合できなかった行も出力され、つまり完全外部結合になる。

その下の AWK は、例示した SQL 文では中間（2 番目、最も外側の SELECT 文から 1 つ内側）の SELECT 文に相当する。つまり完全外部結合した結果、たくさん生じた NULL 値に対する代替値を設定している。会員 ID に関しては NULL でない表のものを採用し、回数に関しては NULL だったら 0 を設定している。

そして次に、sm2 というコマンド（詳細は、このコマンドを説明している HACK 10.45 を参照）を使っている。もちろんここは AWK コマンドを使い、1 列目〜3 列目が同じ値が続く限り 4 列目を足し合わせ、合計値を得るという記述をしてもいいのだが、コードが増えて読みづらくなってしまう。そこで同じことをこなせるこのコマンドを利用した。この例における “sm2 1 3 4 4” とは、「1 列目から 3 列目までの値がすべて同一の行の、4 列目から 4 列目（つまり 4 列目）の合計値を求める」という動作をする。最後に、その結果をもって操作回数と会員 ID でソートを掛けている。

では次に、アクセスログ表ファイルの行数が膨大で事前ソートが困難な場合の別解を記す。

■UNIX – 事前ソート困難で AWK コマンドを使う場合

```
# 例．アクセスログ表ファイル"ACCESSLOG.txt"（会員ID列でソート可能な規模とする）
#       1列目．日時列
#       2列目．会員ID列
#       3列目．操作内容
#     と、会員表ファイル"MEMBERS.txt"（会員ID列でソート済とする）
#       1列目．会員ID列
#       2列目．苗字（漢字）列
#       3列目．名前（漢字）列
#                 :
#     の会員ID列を完全外部結合し、同一会員ID列値の出現回数を集計
#     ただし、会員表由来でアクセスログ表になかった行の出現回数は0とする
#     最終的に出力する列の構成は次の通りとし
#       1列目．会員ID列
#       2列目．操作回数列
#       3列目．操作回数列
#       4列目．操作回数列
#     回数の多い順→会員ID順でソートして出力
#
#     ※ コマンドの間にあるコメント行（"1:日時 2:会員ID"など）は、
#        その位置を流れるデータの列構成を表している
#     ※ 列番号の"NF"とは最終列番号を意味している
#
cat "ACCESSLOG.txt"                                                    |
self 1                                                                 |
```

```
# 1:会員ID                                                          #
awk 'BEGIN{ # 会員表ファイルある全会員IDと氏名を記憶                  #
            while (getline < "MEMBERS.txt") {name[$1]=$2 " " $3;} }  #
                                                                     #
     {      # アクセスログ表由来の行の会員IDの数を集計する          #
            times[$1] = ($1 in times) ? times[$1]+1 : 1;          }  #
                                                                     #
     END  { # アクセスログ表由来の会員IDと操作回数を列挙する        #
            for (id in times) {                                      #
              print id, (id in name) ? name[id]:"* *", times[id];    #
            }                                                        #
            # 会員表のみに由来する会員IDと操作回数(=0)を列挙する     #
            for (id in name) {                                       #
              if (id in times) {continue;}                           #
              print id,                name[id]      ,         0;    #
            }                                                   }' |
# 1:会員ID 2:苗字 3:名前 4:回数(集計済)                             #
sort -b -k 4nr,4 -k 1,1
```

大きな AWK スクリプトに力任せなコードが出来上がり、どの部分が元の SQL 文のどの句に相当するのかがわかりにくくなってしまったが致し方ない。完全外部結合を担う AWK、集計をする AWK（あるいは sm2 コマンド）のように、役割を分解することもできるのだが、sort コマンドを使う回数が増え、ソートを避けるという趣旨から逸脱するのでこのようなコードを別解とした。

このように、AWK コマンドは万能であり、AWK コマンドさえあればそれだけで多くの SQL 文が翻訳できてしまう。だからといって多用することは避けたいが……。（何より可読性が低下し、後々のメンテナンスをやりづらくしてしまう）

HACK 9.5　CROSS OUTER JOIN 句

CROSS OUTER JOIN 句は、2 つの表を交差結合するためのものである。すなわち、片方の表 A のある 1 行を、もう片方の表 B の各行に結合した行を作る。これを表 A のすべての行に対して行う。したがって、表 A の行数が m、表 B の行数が n であるなら、結合後の行数は $m \times n$ である。

SQL 文での使用例を示す。

■SQL

```
-- 例1. 1桁の数字1～9が入った表"NUMBERS"があり、
--       1列目. 1桁数字列"number"
--      これを二重に用意して交差結合し、次のような九九の表を出力
--       1列目. 掛けられる数の列"a"
--       2列目. 掛ける数の列"b"
--       3列目. 積"product"
--      ただし、1の段から順に9の段まで順番に表示
SELECT   A."number"              AS "a",
```

```
        B."number"              AS "b",
        A."number" * B."number" AS "product"
FROM    (SELECT "number" FROM "NUMBERS") AS A
          CROSS OUTER JOIN
        (SELECT "number" FROM "NUMBERS") AS B
ORDER BY A."number" ASC,
        B."number" ASC;

-- 例2. 会員表"MEMBERS"
--       1列目. 会員ID列"id"
--       2列目. 苗字（漢字）列"Myoji"
--       3列目. 名前（漢字）列"Namae"
--                :
--      の1列目～3列目による部分表と、年会費額表"ANNUALFEES"
--       1列目. 年度列"year"
--       2列目. その年の会費額列"fee"
--      を交差結合し、次の列構成の表を出力
--       1列目. 年度列"year"
--       2列目. 会員ID列"id"
--       3列目. 苗字（漢字）列"Myoji"
--       4列目. 名前（漢字）列"Namae"
--       5列目. その年の会費額列"fee"
--      ただし、年度列→会員ID列の順に昇順ソートされた状態にする
SELECT   FEE."year"  AS "year" ,
         MEM."id"    AS "id"   ,
         MEM."Myoji" AS "Myoji",
         MEM."Namae" AS "Namae",
         FEE."fee"   AS "fee"
FROM     (SELECT "id","Myoji","Namae" FROM "MEMBERS"   ) AS MEM
           CROSS OUTER JOIN
         (SELECT "year","fee"         FROM "ANNUALFEES") AS FEE
ORDER BY FEE."year" ASC,
         FEE."id"   ASC;
```

これらを UNIX の世界に翻訳すると次のようになる。

■UNIX

```
# 例1. 1桁の数字1～9が入った表ファイル"NUMBERS.txt"（ソート済）
#       1列目. 1桁数字列
#      があり、これを二重に用意して交差結合し、次のような九九の表を出力
#       1列目. 掛けられる数の列
#       2列目. 掛ける数の列
#       3列目. 積
```

```
#      ただし、1の段から順に9の段まで順番に表示
cat "NUMBERS.txt" | while read a; do
  cat "NUMBERS.txt" | while read b; do
    echo $a $b $((a*b))
  done
done

# 例2. 会員表ファイル"MEMBERS.txt"
#       1列目. 会員ID列
#       2列目. 苗字（漢字）列
#       3列目. 名前（漢字）列
#                :
#      の1列目～3列目による部分表と、年会費額表"ANNUALFEES.txt"
#       1列目. 年度列
#       2列目. その年の会費額列
#      を交差結合し、次の列構成の表を出力
#       1列目. 年度列
#       2列目. 会員ID列
#       3列目. 苗字（漢字）列
#       4列目. 名前（漢字）列
#       5列目. その年の会費額列
#      ただし、年度列→会員ID列の順に昇順ソートされた状態にする
#
#     ※ コマンドの間にあるコメント行（"1:年度 2:会員ID"など）は、
#        その位置を流れるデータの列構成を表している
#
awk 'BEGIN{                                                    #
       while (getline < "MEMBERS.txt") {                       #
         id=$1; myoji=$2; namae=$3;                            #
         while (getline < "ANNUALFEES.txt") {                  #
           print $1,id,myoji,namae,$2;                         #
         }                                                     #
         close("ANNUALFEES.txt"); # closeを忘れないこと        #
                                  #（忘れると外側のループは1周で終わる）#
       }                                                       #
     }'                                                        |
# 1:年度 2:会員ID 3:苗字 4:名前 5:会費額                       #
sort -b -k 1,1 -k 2,2
```

ポイントは、ファイルの行ループを二重にすることである。例 1 ではシェルスクリプトで実装し、例 2 では AWK スクリプトで実装している。例 1 の方も AWK で実装したいところだが、AWK スクリプトでは同一ファイルを多重に開けないのでシェルスクリプトで実装している。

第 10 章

SQL to UNIX マイグレーション (4) – 関数

SQL to UNIX マイグレーションの最後の章では関数を扱う。

たくさんあり、かつ製品によってバラバラなものもあってすべてを網羅できるわけではないが、主要なものは押えておいたつもりだ。すべてのものは網羅できていなくとも、UNIX へのマイグレーションはどんな感じでできるのかということが伝わることを願う。

なお、この章でも POSIX 標準の UNIX コマンドのみならず、HACK 1.7 で紹介した “ShellShoccar” コマンドセット（Tukubai コマンド含む）が登場するので、その HACK を参照して準備しておくこと。

HACK 10.1　ABS 関数

ABS 関数は、絶対値を求める関数である。

SQL 文の例を示す。

■SQL

```
-- 例．-201の値を出力（答えは201）
SELECT ABS(-201);
```

UNIX の世界では、AWK コマンドを使えばよいが、絶対値を求める関数はない。しかし三項演算子を使って、負値であったら −1 を掛けるという演算をすればいいだけだ。

■UNIX

```
# 例．-201の値を出力（答えは201）
echo -201                              |
awk '{print ($1>=0) ? $1 : -$1;}'
```

HACK 10.2　ACOS 関数

ACOS 関数は、アークコサイン値を求める関数である。戻り値の単位はラジアンである。

SQL 文の例を示す。

■SQL

```
-- 例. arccos(0.5)の値を出力（高精度であれば答えは約1.04719755119660）
SELECT ACOS(0.5);
```

UNIX の世界では AWK コマンドを用いるのが最も簡単ではあるものの、arccos に相当するものはない。しかし、arccos は

$$\arccos x = \arctan \frac{\sqrt{1-x^2}}{x} \tag{10.1}$$

という式で求められるので、AWK の atan2 関数と sqrt 関数の組み合わせで得られる。

■UNIX

```
# 例. arccos(0.5)の値を出力（高精度であれば答えは約1.04719755119660）
echo 0.5                                        |
awk 'BEGIN{OFMT="%.14f";                      }  #
     {     print atan2(sqrt(1-$1*$1)/$1,1);}'

#  （arccos計算を関数化した別解）
echo 0.5                                                 |
awk 'BEGIN{OFMT="%.14f";                               }  #
     {     print arccos($1);                           }  #
     function arccos(x) {return atan2(sqrt(1-x*x)/x,1);}'
```

AWK のデフォルト精度は小数点以下 6 桁だが、BEGIN セクションなどで “OFMT="%.14f"” と設定しておけば精度を 14 桁に上げられる。

HACK 10.3 ASIN 関数

ASIN 関数は、アークサイン値を求める関数である。戻り値の単位はラジアンである。

SQL 文の例を示す。

■SQL

```
-- 例. arcsin(0.5)の値を出力（高精度であれば答えは約0.52359877559830）
SELECT ASIN(2);
```

UNIX の世界では AWK コマンドを用いるのが最も簡単ではあるものの、arcsin に相当するものはない。しかし、arcsin は

$$\arcsin x = \arctan \frac{x}{\sqrt{1-x^2}} \tag{10.2}$$

という式で求められるので、AWK の atan2 関数と sqrt 関数の組み合わせで得られる。

■UNIX

```
# 例. arcsin(0.5)の値を出力（高精度であれば答えは約0.52359877559830）
echo 0.5                                          |
```

```
awk 'BEGIN{OFMT="%.14f";                    }  #
     {     print atan2($1/sqrt(1-$1*$1),1);}'

#  (arcsin計算を関数化した別解)
echo 0.5                                                  |
awk 'BEGIN{OFMT="%.14f";                                }  #
     {     print arcsin($1);                            }  #
     function arcsin(x) {return atan2(x/sqrt(1-x*x),1);}'
```

AWK のデフォルト精度は小数点以下 6 桁だが、BEGIN セクションなどで “`OFMT="%.14f"`” と設定しておけば精度を 14 桁に上げられる。

HACK 10.4　ATAN / ATAN2 関数

ATAN 関数は、アークタンジェント値を求める関数である。ATAN2 関数も同様であるが、引数 2 つで指定する点が異なる。両者の関係は “`ATAN(y/x)=ATAN2(y,x)`” である。

SQL 文の例を示す。

■SQL

```
-- 例1. arctan(2)の値を出力 (高精度であれば答えは約1.10714871779409)
SELECT ATAN(2);

-- 例2. arctan(2/1)の値を出力 (高精度であれば答えは約1.10714871779409)
SELECT ATAN2(2,1);
```

UNIX の世界では AWK コマンドを用いるのが最も簡単ではあるものの、存在するのは atan2 の方だけである。もし arctan の引数として 1 つのみが与えられているなら、第 2 引数には 1 を与えればよい。よって次のように翻訳される。

■UNIX

```
# 例1. arctan(2)の値を出力 (高精度であれば答えは約1.10714871779409)
echo 2                            |
awk 'BEGIN{OFMT="%.14f";      }  #
     {     print atan2($1,1);}'

# 例2. arctan(2/1)の値を出力 (高精度であれば答えは約1.10714871779409)
echo 2 1                           |
awk 'BEGIN{OFMT="%.14f";       }  #
     {     print atan2($1,$2);}'
```

AWK のデフォルト精度は小数点以下 6 桁だが、BEGIN セクションなどで “`OFMT="%.14f"`” と設定しておけば精度を 14 桁に上げられる。

HACK 10.5 AVG 関数

SUM 関数は、GROUP BY 句と共に用いる集約関数の一種で、集約される列の平均値を返す。
SQL 文の例を示す。

■SQL

```
-- 例．テストの成績表"EXAMSCORES"
--      1列目．学生ID列"id"
--      2列目．学期（1-3）列"term"
--      3列目．教科名列"subject"
--      4列目．点数列"score"
--     で、各学期・各教科の平均点を出力
--     列の構成は次の通り
--      1列目．学期（1-3）列"term"
--      2列目．教科名列"subject"
--      3列目．平均点"averagescore"
SELECT   "id"         AS "term"        ,
         "term"       AS "subject"     ,
         AVG("score") AS "averagescore"
FROM     "EXAMSCORES"
GROUP BY "term"   ,
         "subject";
```

UNIX の世界に翻訳するには、まずは処理を少しでも軽くするために、不要な列（学生 ID）列は削除する。そのうえで同じ学期、かつ同じ教科が隣り合うように sort コマンドを掛け、それを AWK コマンドに与えて同じ学期・教科が続く限り点数を足し合わせて最後に平均を求める。

この方針で記述すると次のようになる。

■UNIX

```
# 例．テストの成績表ファイル"EXAMSCORES.txt"
#      1列目．学生ID列
#      2列目．学期（1-3）列
#      3列目．教科名列
#      4列目．点数列
#     で、各学期・各教科の平均点を出力
#     列の構成は次の通り
#      1列目．学期（1-3）列
#      2列目．教科名列
#      3列目．平均点
cat "EXAMSCORES.txt"                                                    |
self 2 3 4                                                              |
# 1:学期 2:教科 3:点数                                                  #
```

```
sort -b -k 2,2 -k 3,3                                                    |
awk 'BEGIN                  {# キー文字列,合計点,人数を初期化              #
                             if (getline) {prev_keys=$1 " " $2;total=$3;n=1;    #
                             } else       {prev_keys=""; exit;              }}  #
                                                                          #
     $1 " " $2 != prev_keys{print prev_keys,total/n; # キーの平均点を出力      #
                             prev_keys=$1 " " $2;     # 新たなキーをセット      #
                             total=0;                 # スコアをリセット        #
                             n    =0;                 # 人数をリセット       } #
                                                                          #
     {                       total+=$3; n++; # 点数を加算し、人数を+1       } #
                                                                          #
     END                    {# 未出力のキーの平均点を出力                   #
                             if (prev_keys=="") {exit;}                   #
                             print prev_keys,total/n;                    }'
```

HACK 10.6　CAST / CONVERT 関数（データ型変換）

CAST、または CONVERT（製品によって名前や書式が異なる）は、SQL で取り扱う表が持つデータ型を変換するための関数である。

製品によって方言が激しいが、とりあえず SQL 文の例を示す。（REPLACE 関数については HACK 10.38 参照）

■SQL

```
-- 例．表"MEMBERS"
--      1列目．ID列"id"（文字列型、0埋めされた5桁の数字から成る）
--      2列目．会員証発行回数列"times"（数値型、1から始まる）
--      3列目．入会年月日列"entdate"（時刻型、"YYYY-MM-DD"形式）
--     を、次のように修正して出力
--      1列目．ID列"id"（数値化して100,000を足す）
--      2列目．会員証「再」発行回数列"times"（1を引いて、"回"を付け、文字列化）
--      3列目．入会年月日列"entdate"（"YYYY/MM/DD"表記にするため文字列化）
SELECT         CAST( "id"       AS NUMBER  ) + 100000 AS "id"    ,
               CAST(("times"-1) AS VARCHAR2) || '回'  AS "times",
       REPLACE(CAST( "entdate"  AS VARCHAR2),'-','/') AS "times",
FROM   "MEMBERS";
```

UNIX の世界（テキスト処理の世界）ではもともとデータ型という概念が無く、敢えて言うならすべて可変長の文字列型である。ただ、AWK コマンドに取り込んだ場合には、数値型か文字列型かが暗黙的（動的）に決まる。“0”～“9”、それに “+”、“-”、“.” などから成る数字と解釈でき得る文字列は、気を付けないと意図しない方の型になる恐れがある。そこでここでは、上記の例を元に、AWK コマンド上の変数を意図的に数値型あるいは文字列型に設定する方法を示す。

■UNIX

```
# 例. 表ファイル"MEMBERS.txt"
#      1列目. ID列（0埋めされた5桁の数字から成る）
#      2列目. 会員証発行回数列（1から始まる）
#      3列目. 入会年月日列（"YYYY-MM-DD"形式）
#     を、次のように修正して出力
#      1列目. ID列（元の数に100,000を足す）
#      2列目. 会員証「再」発行回数列（1を引いて、"回"を付ける）
#      3列目. 入会年月日列"entdate"（"YYYY/MM/DD"表記にする）
cat "MEMBERS.txt"                                          |
awk '{id      = ($1*1) + 100000;                           #
      times   = ($2-1)   "回"  ;                           #
      entdate = $3             ; gsub(/-/,"/",entdate);    #
      print id, times, dntdate;                          }'
```

AWK コマンドで暗黙的に決まる数値型・文字列型を明確に決めるには、次のような、内容の変化しない操作を介在させるとよい。

- 確実に数値型にする
  - 0 を足す“(var+0)”
  - 1 を掛ける“(var*1)”
- 確実に文字列型にする
  - 空文字を結合する“(var "")”

この丸括弧の外側に数値または文字列としての意図した操作をするのが安全だ。これをしないと場合によっては、数値型だと思って 2 を掛けたのに、実は文字列型（値としては 0 扱い）として扱われていてたために 0 になったなどの事故が起こる。

UNIX の世界（テキスト処理の世界）ではこの点にだけ気を付ければ後は型変換を意識する必要はないと思われる。ゆえに、上記例の日付データに関しては最初から最後まで文字列として扱っている。

HACK 10.7 CEIL / CEILING 関数

CEIL または CEILING 関数（製品によって名前が異なる）は、与えられた数値の切り上げを行う関数である。製品によっては、第 2 引数 n を指定して小数点第 n 位に切り上げる（1 を指定した場合、1.09 が 1.1 になる）こともできる。

SQL 文の例を示す。

■SQL

```
-- 例1. 1.58の切り上げ値（整数）を出力（答えは2）
SELECT CEIL(1.58);

-- 例2. -1.58の切り上げ値（小数第1位まで）を出力（答えは-1.5）
SELECT CEIL(-1.58,1);
```

UNIX の世界では、AWK コマンドの int 関数を使って（端数があれば）切り捨てた後に +1……と言い

たいところだが、この int は絶対値方向に切り上げるため、負値で結果が異なる。そこで AWK のユーザー関数として同名関数を用意した。あとは必要な時にこのコード（スニペット）を流用すればよい。

■UNIX

```
# 例1. 1.58の切り上げ値（整数）を出力（答えは2）
echo 1.58 |
awk '{print ceil($1);}
     function ceil(f,n) {
       n=n*1;
       f=f*10^n;
       if (int(f)==f) {return      f    /10^n;}
       if (f > 0    ) {return (int(f)+1)/10^n;}
                       return  int(f)   /10^n;
     }'

# 例2. -1.58の切り上げ値（小数第1位まで）を出力（答えは-1.5）
echo -1.58 1 |
awk '{print ceil($1,$2);}
     function ceil(f,n) {
       n=n*1;
       f=f*10^n;
       if (int(f)==f) {return      f    /10^n;}
       if (f > 0    ) {return (int(f)+1)/10^n;}
                       return  int(f)   /10^n;
     }'
```

HACK 10.8　COALESCE 関数

COALESCE 関数は、NULL 値を非 NULL 値に置き換える関数の一種で、引数として与えられた値を第 1 引数から順番に検査し、非 NULL 値である最初の引数の値を返すという関数である。（引数がすべて NULL 値だった場合には NULL 値を返す）

NULL 値を非 NULL 値に置き換える関数として他に、IFNULL、ISNULL、NVL といったものがあるが、それらはどれも、この関数の引数が 2 つの場合と見なせる。

SQL 文の例を示す。

■SQL

```
-- 例．表"MY_TBL"の"a"列、"b"列、"c"列の中から、NULL値でない列の値を1つ出力
--      ただし、非NULL値が複数あった場合、優先順位は"a"列、"b"列、"c"列の順とし、
--      すべてNULL値であった場合には数値として‘‘\jverb|0|’’を出力
SELECT COALESCE("a","b","c",0) FROM "MY_TBL";
```

UNIX（テキストデータ）の世界では、HACK 2.4 で解説したように（直接的な意味での）NULL 値というものが存在しない。従って NULL 値を表現するためには、そこの HACK で取り決めたように “-” や

“*” などの 1 文字を割り当てていたはずだ。

ゆえに COALESCE 関数の翻訳とは、文字列がそれらの内容だった場合には次の候補を割り当てるという動作に翻訳される。そこで CASE 句（HACK 8.14）の UNIX 側の例と同様に三項演算子（または if 文）を用いて次のように翻訳できる。

■UNIX

```
# 例. 表ファイル"MY_TBL.txt"の2列目、3列目、4列目の中から、
#     NULL値（"-"とする）でない列の値を1つ出力
#     ただし、非NULL値が複数あった場合、優先順位は2列目、3列目、4列目の順とし、
#     すべてNULL値であった場合には数値として‘‘\jverb|0|’’を出力
cat "MY_TBL.txt"               |
awk ’{print ($2!="-") ? $2 :   \
            ($3!="-") ? $3 :   \
            ($4!="-") ? $4 :   \
                        0 ;}’

# （三項演算子を用いない別解）
cat "MY_TBL.txt"                        |
awk ’{if        ($2!="-") {ret=$2;    #
      } else if ($3!="-") {ret=$3;    #
      } else if ($4!="-") {ret=$4;    #
      } else              {ret= 0;}   #
      print ret;                    }’
```

なお、NULL 値を非 NULL 値に置き換えるという処理の実例については、FULL OUTER JOIN 句の解説（HACK 9.4）を参照。

HACK 10.9 CONCAT 関数

CONCAT 関数は、文字列を結合して 1 つの文字列を返す関数である。

SQL 文の例を示す。

■SQL

```
-- 例. 会員表"MEMBERS"の2列目（苗字列"Myoji"）と3列目（名前列"Namae"）を、
--     1つの半角空白を挿んで結合し、氏名列"name"とする
--     列の構成は次の通り
--      1列目. 会員ID列"id"（元表のまま）
--      2列目. 氏名列"name"
SELECT   "id"                         AS "id"   ,
         CONCAT("Myoji",’ ’,’Namae’) AS "name"
FROM     "MEMBERS";
```

UNIX の世界では、文字列過去と捉えて sed コマンドで処理する方法もあるが、多くの場合に簡単かつ素直に書けるのは AWK コマンドを使う方法だろう。そして AWK コマンドの場合、文字列の結合は関数

ではなく演算子を使う。ここまで本書では明記してこなかったが、AWK の文字列結合演算子は“ ”（半角空白）である。

なお、HACK 2.2 で取り決めたように、UNIX の世界では半角空白はアンダースコア“_”で扱うと決めているので、ここでの苗字と名前の結合においても、間に挿む文字は半角空白の代わりにアンダースコアとする[*1]。

■UNIX

```
# 例．会員表ファイル"MEMBERS.txt"の2列目（苗字列）と3列目（名前列）を、
#     1つの半角空白（アンダースコア"_"で代用）を挿んで結合し、氏名列とする
#     列の構成は次の通り
#      1列目．会員ID列（元表のまま）
#      2列目．氏名列
awk '{print $1, $2 "_" $3;}' "MEMBERS.txt"

# （sedコマンドで行う場合の別解）
cat "MEMBERS.txt"                                                              |
sed 's/^\([^ ]\{1,\} \{1,\}\)\([^ ]\{1,\}\) \{1,\}\([^ ]\{1,\}\).*/\1\2_\3/'
```

一応 sed コマンドを用いた例も書いたが、このようにして AWK と比べるとだいぶ分かりにくくなってしまうことが見て取れる。

HACK 10.10　CONVERT 関数（データ型変換）

→ CAST 関数（HACK 10.6）参照

HACK 10.11　CONVERT 関数（文字コード変換）

CONVERT 関数は、文字コード変換をするための関数またはデータ型変換をするための関数（製品によってどちらかの機能を持つ）であるが、ここでは前者について説明する。

引数を 2 つまたは 3 つとり、第 1 引数は変換対象の文字列、第 2 引数は変換先の文字コード、第 3 引数は変換元の文字コードである。

SQL 文の例を示す。

■SQL

```
-- 例．表"MEMBERS"
--      1列目．ID列"id"
--      2列目．苗字列"Myoji"
--      3列目．名前列"Namae"
--      4列目．補足列"description"
--     のうちの、1列目以外は全角文字が使われており、文字コードはShift-JISである
--     これをUTF-8に変換して上書き保存
```

[*1] この例では苗字や名前にアンダースコアは無いという前提であるが、あり得る場合については HACK 2.2 を参照

```
UPDATE "MEMBERS"
SET    "Myoji"       = CONVERT("Myoji"      ,'AL32UTF8','JA16EUCTILDE'),
       "Namae"       = CONVERT("Namae"      ,'AL32UTF8','JA16EUCTILDE'),
       "description" = CONVERT("description",'AL32UTF8','JA16EUCTILDE');
```

UNIX の世界に翻訳する場合、特定の列だけ文字コード変換するうまい方法は無い。表データ（ファイル）全体をある文字コードからある文字コードへ一気に変換する方法はあり、そのためには iconv コマンドを使う。

なお、HACK 4.5 で説明した通り、File/Dir Hack においては元の表ファイルに書き戻さずに、変換したデータは別ファイルに書き込む。

以上を踏まえて、UNIX の世界に翻訳したコードを記す。

■UNIX

```
# 例. 表ファイル"MEMBERS.txt"
#      1列目. ID列"id"
#      2列目. 苗字列"Myoji"
#      3列目. 名前列"Namae"
#      4列目. 補足列"description"
#     のうちの、1列目以外は全角文字が使われており、文字コードはShift-JISである
#     これをUTF-8に変換して、新たな表ファイル"MEMBERS2.txt"に書き込む
iconv -cs -f CP932 -t UTF-8 "MEMBERS.txt" > "MEMBERS2.txt"
```

一般的に “Shift-JIS” と呼ばれている文字コードは、純粋な Shift-JIS ではなく、“CP932” という形式である場合が多い。ゆえにここでは変換元のコードしてそれを指定している。また、変換先文字コードに対応する文字が存在せずに部分的に変換に失敗することがあるので、それでも警告無しに変換を進めさせるため、“-s” と “-c” というオプションを併用している。

HACK 10.12 COS 関数

COS 関数は、余弦（コサイン）値を求める関数である。引数の単位はラジアン。

SQL 文の例を示す。

■SQL

```
-- 例. cos(pi)の値を出力（高精度であれば答えは-1）
SELECT COS(3.14159265358979);
```

UNIX の世界では、AWK コマンドの cos 関数がこれに相当するので、それを使えばいい。

■UNIX

```
# 例. cos(pi)の値を出力（高精度であれば答えは-1）
echo 3.14159265358979       |
awk 'BEGIN{OFMT="%.14f"; }  #
     {     print cos($1);}'
```

AWK のデフォルト精度は小数点以下 6 桁だが、BEGIN セクションなどで “OFMT="%.14f"” と設定しておけば精度を 14 桁に上げられる。

HACK 10.13　CURRENT_DATE / CURRENT_TIME / CURRENT_TIMESTAMP 関数

CURRENT_DATE、CURRENT_TIME、CURRENT_TIMESTAMP（製品によって多少の方言あり）といった関数は、SQL 文実行時の現在年月日（年月日のみ）や現在時刻（時分秒のみ）、現在日時（年月日時分秒）を返す関数である。関数ではあるが引数を取らない[*2]ので、SQL 文では丸括弧無しで組み込み変数のように使える製品が多い。

SQL 文の使用例を示す。

■SQL

```
-- 例1．現在日付（年月日）を出力
SELECT CURRENT_DATE AS "time";

-- 例2．現在日時（年月日時分秒）を出力
SELECT CURRENT_TIMESTAMP AS "time";
```

UNIX の世界で現在時刻を持っているものといえば、date コマンド（詳細は各種 OS にある date の man ページ参照）である。従って次のように翻訳される。

■SQL

```
# 例1．現在日付（年月日）を出力（フォーマットは'YYYY-MM-DD'とする）
date '+%Y-%m-%d'

# 例2．現在日時（年月日時分秒）を出力（フォーマットは'YYYYMMDDhhmmss'とする）
date '+%Y%m%d%H%M%S'

# 例2の現在時刻を太平洋標準時（PST）で出力したい場合
TZ='PST-8'date '+%Y%m%d%H%M%S'

# 例2の現在時刻をUTC時刻で出力したい場合
date -u '+%Y%m%d%H%M%S'

# 例2の現在時刻をAWKの中で使いたい場合
t=$(date '+%Y%m%d%H%M%S')
awk -v time=$t '{timeという名前の変数に入ったので...煮るなり焼くなり...}'

# 例2の現在時刻をAWKの中で使いたい場合（別解）
awk 'BEGIN{time='"$(date '+%Y%m%d%H%M%S')"';}
     {     timeという名前の変数に入ったので...煮るなり焼くなり...}'
```

*2 「ミリ秒単位まで」のような精度を引数で与えられる製品もある。

date コマンド自体は標準出力に値を出力するだけだが、AWK などのコマンドに渡す方法も示したので他のコマンドからの使い方もこれでわかるだろう。また、様々なフォーマットやタイムゾーンで出力する方法も示したので、これで国際対応もできるだろう。

10.13.1 日時・時間の加減算はどうやるのか

SQL の世界における日時・時間という値は、それ専用の型で取り扱うため、ユーザーは何も意識せず自然に加減算ができる。しかし、UNIX の世界（というよりテキスト処理の世界）では型といえばせいぜい数値か文字列かしかないので、日時計算が難しい。日時を例えば “YYYYMMDDhhmmss” のような 14 桁の数字で表したところで単純な加減算はできない。なぜなら、年、月、日、時、分、秒それぞれの単位で繰り上がりの数字が異なるからだ。

まずは SQL で当たり前に行える例を 1 つ示す。

■SQL

```
-- 例．操作ログ表"OPLOG"
--       1列目．操作開始日時"starttime"（時刻型）
--       2列目．操作終了日時"endtime"（時刻型）
--       3列目．操作内容"description"
--      で、操作時刻の後ろに操作日時から現在までの経過時間列と、
--      同じ操作を今からした場合のを挿入して出力
--      列の構成は次の通り
--       1列目．操作開始日時"starttime"（時刻型）
--       2列目．操作終了日時"endtime"（時刻型）
--       3列目．所要時間"duration"
--       4列目．現在時刻から行った場合の予想終了時刻"exp_endtime"
--       5列目．操作内容"description"
SELECT   "starttime"         AS "starttime"          ,
         "endtime"         AS "endtime"              ,
         "endtime" - "starttime" AS "duration"       ,
         CURRENT_TIMESTAMP + "endtime" - "starttime",
         "description" AS "description"
FROM     "OPLOG";
```

これを UNIX の世界に翻訳すると、次のようになる。

■UNIX

```
# 例．操作ログ表ファイル"OPLOG.txt"
#       1列目．操作開始日時（ISO8601形式、"YYYY-MM-DDThh:mm:ss"）
#       2列目．操作終了日時（ISO8601形式、"YYYY-MM-DDThh:mm:ss"）
#       3列目．操作内容"description"
#      で、操作時刻の後ろに操作日時から現在までの経過時間列と、
#      同じ操作を今からした場合のを挿入して出力
#      列の構成は次の通り
```

```
#       1列目. 操作開始日時（同形式）
#       2列目. 操作終了日時（同形式）
#       3列目. 所要時間（秒数）
#       4列目. 現在時刻から行った場合の予想終了時刻（同形式）
#       5列目. 操作内容
#
#     ※ コマンドの間にあるコメント行（"1:開始時 2:終了時"など）は、
#        その位置を流れるデータの列構成を表している
#
cat "OPLOG.txt"                                                          |
# 1:開始時 2:終了時 3:操作内容                                           #
calclock 1 2                                                             |
# 1:開始時 2:開始時(UNIX時) 3:終了時 4:終了時(UNIX時) 5:操作内容         #
awk 'BEGIN{now_in_unix='"$(date '+%Y%m%d%H%M%S' | calclock 1 | self 2)";'}  #
     {     print $1, $3, $4-$2, $4-$2+now_in_unix, $5;                 }' |
# 1:開始時 2:終了時 3:経過秒数 4:予想終了時(UNIX時) 5:操作内容           #
calclock -r 4                                                            |
# 1:開始時 2:終了時 3:経過秒数 4:予想終了時(UNIX時) 5:予想終了時 6:操作内容 #
self 1 2 3 5 6                                                           |
# 1:開始時 2:終了時 3:経過秒数 4:予想終了時 5:操作内容
```

UNIX の世界（しつこいようだがテキスト処理の世界）ではどうすればいいのかというと、UNIX 時間（1970-01-01T00:00:00+UTC からの経過秒数）に変換したうえで加減算を行い、最後にカレンダー日時（YYYYMMDDhhmmss）に逆変換すればよい。この UNIX 時間との相互変換を行うコマンドが Tubukai コマンドセットの中にあって（POSIX 版としても移植済）、それは calclock[*3] と呼ばれている。

“`calclock` *a* [*b c* ...]” という引数で実行すると、*a* 列目（*b* 列目…）それぞれの文字列をカレンダー日時（YYYYMMDDhhmmss）と見なしたうえで、それらを UNIX 時間に変換したものをそれぞれの右隣の列に挿入する（元の列は残したまま）。“`calclock` -r *a* [*b c* ...]” という引数で実行すると、*a* 列目（*b* 列目…）それぞれの文字列を UNIX 時間とみなしたうえで、それらをカレンダー日時（YYYYMMDDhhmmss）に変換したものをそれぞれの右隣の列に挿入する（元の列は残したまま）。

このコマンドを使い、元列の開始日時、終了日時それぞれの UNIX 時間を得たうえで AWK に渡している。そして、AWK の BEGIN セクションでは予め現在時刻の UNIX 時間[*4]を計算しておき、各行のループの際に、経過秒数と予想終了時刻（UNIX 時間）を計算している。AWK が終わったら、UNIX 時間で表現されている予想終了時刻をカレンダー日時に変換（正確には変換した値を右隣に挿入し、元列を除去）している。

[*3] 詳細は次の URL を参照。
`https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_calclock`

[*4] date コマンドが “`%s`” に対応していれば calclock コマンド無しに UNIX 時間が直接得られるが、POSIX の範囲を逸脱しているのでこのようにした。ここで紹介している calclock コマンドは POSIX 移植版である。

HACK 10.14 DATEDIFF / DATADIFF_BIG 関数

→ 2 つの日時の差を求める関数であるため、「日時・時間の加減算はどうやるのか」（10.13.1 項）参照。

HACK 10.15 DBMS_RANDOM パッケージ

DBMS_RANDOM パッケージは Oracle において乱数を得るためのものである。これは“DBMS_RANDOM.VALUE”と記述した時に、他の RDBMS 製品の RAND や RANDOM 関数と同じ動作をするため、HACK 10.36 を参照。

HACK 10.16 EOMONTH 関数

→ LAST_DAY 関数のところでまとめて解説しているため、HACK 10.22 参照。

HACK 10.17 EXP 関数

EXP 関数は、自然対数を底としたべき乗を求める関数である。
SQL 文の例を示す。

■SQL

```
-- 例．自然対数の底eの2.5乗を出力（高精度であれば答えは12.18249396070347）
SELECT EXP(2.5);
```

UNIX の世界では AWK コマンドにある exp 関数がこれに相当するので、それを使えばいい。

■UNIX

```
# 例．自然対数の底eの2.5乗を出力（高精度であれば答えは12.18249396070347）
echo 2.5                    |
awk 'BEGIN{OFMT="%.14f"; }  #
     {     print exp($1);}'
```

AWK のデフォルト精度は小数点以下 6 桁だが、BEGIN セクションなどで“OFMT="%.14f"”と設定しておけば精度を 14 桁に上げられる。

HACK 10.18 FLOOR 関数

FLOOR 関数は、与えられた数値の切り捨てを行う関数である。製品によっては、第 2 引数 n を指定して小数点第 n 位に切り捨てる（1 を指定した場合、1.09 が 1.0 になる）こともできる。
SQL 文の例を示す。

■SQL

```
-- 例1．1.58の切り捨て値（整数）を出力（答えは1）
```

```
SELECT FLOOR(1.58);

-- 例2. -1.58の切り捨て値（小数第1位まで）を出力（答えは-1.6）
SELECT FLOOR(-1.58,1);
```

UNIX の世界では、AWK コマンドの int 関数を（端数があれば）使えばいい……と言いたいところだが、この int は絶対値方向に切り捨てるため、負値で結果が異なる。そこで AWK のユーザー関数として同名関数を用意した。あとは必要な時にこのコード（スニペット）を流用すればよい。

■UNIX

```
# 例1. 1.58の切り捨て値（整数）を出力（答えは1）
echo 1.58 |
awk '{print floor($1);}
     function floor(f,n) {
       n=n*1;
       f=f*10^n;
       if (int(f)==f) {return      f    /10^n;}
       if (f > 0    ) {return  int(f)   /10^n;}
                       return (int(f)-1)/10^n;
     }'

# 例2. -1.58の切り捨て値（小数第1位まで）を出力（答えは-1.6）
echo -1.58 1 |
awk '{print floor($1,$2);}
     function floor(f,n) {
       n=n*1;
       f=f*10^n;
       if (int(f)==f) {return      f    /10^n;}
       if (f > 0    ) {return  int(f)   /10^n;}
                       return (int(f)-1)/10^n;
     }'
```

HACK 10.19　IFNULL 関数

→ COALESCE 関数（HACK 10.8）参照

HACK 10.20　INITCAP 関数

INITCAP 関数は、与えられた文字列を返す際、文字列内の英単語の先頭を大文字アルファベットに、単語の 2 文字以降は小文字化してから出力する関数である。

SQL 文の例を示す。

■SQL

```
-- 例. 住所録表"MY_ADDRESSES"
--      1列目. 番号列"number"
--      2列目. 住所1（ローマ字）列"address1"
--      3列目. 住所2（ローマ字）列"address2"
--      4列目. 住所3（ローマ字）列"address3"
--      5列目. 苗字（ローマ字）列"Myoji"
--      6列目. 名前（ローマ字）列"Namae"
--     を出力する際、苗字列と名前列はすべて先頭大文字にする
SELECT "number"          AS "number"  ,
       "address1"        AS "address1",
       "address2"        AS "address2",
       "address3"        AS "address3",
       INITCAP("Myoji")  AS "Myoji"   ,
       INITCAP("Namae")  AS "Namae"
FROM   "MY_ADDRESSES";
```

UNIX の世界で、これに相当する関数ないが、AWK コマンドでユーザー関数として作ることはさほど難しいことではないので同名の関数を作って対応する。

■UNIX

```
# 例. 住所録表ファイル"MY_ADDRESSES.txt"
#      1列目. 番号列
#      2列目. 住所1（ローマ字）列
#      3列目. 住所2（ローマ字）列
#      4列目. 住所3（ローマ字）列
#      5列目. 苗字（ローマ字）列
#      6列目. 名前（ローマ字）列
#     を出力する際、苗字列と名前列はすべて先頭大文字にする
cat "MY_ADDRESSES.txt"                                          |
awk '{print $1,$2,$3,$4,initcap($5),initcap($6);}                #
     function initcap(str0, str1) {                              #
       str1="";                                                  #
       while (length(str0)) {                                    #
         if (match(str0, /[A-Za-z]+/))                         { #
           str1 = str1 substr(str0, 1, RSTART-1);                #
           str1 = str1 toupper(substr(str0,RSTART  ,        1)); #
           str1 = str1 tolower(substr(str0,RSTART+1,RLENGTH-1)); #
           str0 = substr(str0,RSTART+RLENGTH);                   #
         } else                                                  { #
           str1 = str1 str0;                                     #
           break;                                                #
```

```
      }                                                    #
    }                                                      #
    return str1;                                           #
  }'
```

10.20.1　全角アルファベットも先頭大文字化したい場合

この内容については 10.27.1 を参照。

HACK 10.21　ISNULL 関数

→ COALESCE 関数（HACK 10.8）参照

HACK 10.22　LAST_DAY / EOMONTH 関数

LAST_DAY 関数は、引数で与えられた日付に対して月末の日付を返す関数である。類似のものとして EOMONTH 関数が使える製品があるが、引数が 1 つの場合は同じ動作をし、2 つの場合は第 1 引数にその数だけ月を加算（または減算）したうえで同じ計算をする。

SQL 文だとこんな感じになる。

■SQL

```
-- 例1．2020年2月13日の月末日付を出力（答えは'2020-02-29'）
SELECT LAST_DAY('2020-02-13');

-- 例2．2020年2月13日の13か月前の月末日付を出力（答えは'2019-01-31'）
SELECT EOMONTH('2020-02-13',-13);
```

UNIX の世界にはこれらをやってくれる関数は無いものの、AWK コマンドのユーザー関数として作ることはできる。そこで、lastday という名前の関数（EOMONTH のように引数 2 個にも対応）を作った。一度作ってしまえば、あとはこのコード（スニペット）を使い回すだけだ。

■UNIX

```
# 例1．2020年2月13日の月末日付を出力（答えは'20200229'）
#      * 日付は（"YYYYMMDD"または"YYYYMMDDhhmmss"形式で与える）
echo 20200213 |
awk '{print lastday($1);}
     function lastday(date,n ,y,m,t,D) {
       if (! match(date,/^[0-9][0-9][0-9][0-9][0-9][0-9]/)) {return "";}
       n=n*1;
       split("31 0 31 30 31 30 31 31 30 31 30 31",D);
       y=substr(date,1,4)*1; m=substr(date,5,2)*1; t=substr(date,9);
       m=m+n;
       y=y+((m>0)?int((m-1)/12):int((m-12)/12));
```

```
      m=(m%12==0)?12:(m>0)?m%12:m%12+12;
      if (!(m in D)       ) {return                    "";}
      if (D[m]  >  0      ) {return y*10000+m*100+D[m] t;}
      if (y%4   != 0      ) {return y*10000+  200+  28 t;}
      if (y%400 == 0      ) {return y*10000+  200+  29 t;}
      if (y%100 == 0      ) {return y*10000+  200+  28 t;}
                             return y*10000+  200+  29 t;
    }'

# 例2. 2020年2月13日の13か月前の月末日付を出力（答えは'20190131'）
echo 20200213 -13 |
awk '{print lastday($1,$2);}
    function lastday(date,n ,y,m,t,D) {
      if (! match(date,/^[0-9][0-9][0-9][0-9][0-9][0-9]/)) {return "";}
      n=n*1;
      split("31 0 31 30 31 30 31 31 30 31 30 31",D);
      y=substr(date,1,4)*1; m=substr(date,5,2)*1; t=substr(date,9);
      m=m+n;
      y=y+((m>0)?int((m-1)/12):int((m-12)/12));
      m=(m%12==0)?12:(m>0)?m%12:m%12+12;
      if (!(m in D)       ) {return                    "";}
      if (D[m]  >  0      ) {return y*10000+m*100+D[m] t;}
      if (y%4   != 0      ) {return y*10000+  200+  28 t;}
      if (y%400 == 0      ) {return y*10000+  200+  29 t;}
      if (y%100 == 0      ) {return y*10000+  200+  28 t;}
                             return y*10000+  200+  29 t;
    }'
```

HACK 10.23 LCASE 関数

→ LOWER 関数（HACK 10.27）参照

HACK 10.24 LEN / LENGTH / LENGTHB 関数

LEN、LENGTH 関数は、引数として与えられた文字列の長さ（文字数、ただし MySQL ではバイト数）を返す関数である。LENGTHB は文字列の長さを、バイト数で返す関数である。（製品によって使える関数の名前が違う）

■SQL

```
-- 例1. 文字列「円周率は3」の文字数を出力（5が得られる）
--       ※ MySQLではバイト数を返すので注意
```

```
SELECT LENGTH('円周率は3')

-- 例2．文字列「円周率は3」のバイト数を出力（UTF-8環境では13が得られる）
SELECT LENGTHB('円周率は3');
```

UNIX の世界に翻訳するなら、文字列長やバイト数を数えるのは AWK コマンドの length 関数である。ただし、文字列長を数えるかバイト数を数えるかは環境変数（LANG や LC_CTYPE、あるいは LC_ALL）の設定によって変わる。

■UNIX

```
# 例1.文字列「円周率は3」の文字数を出力（文字列がUTF-8の場合、5が得られる）
echo '円周率は3'                                |
LC_ALL=ja_JP.UTF-8 awk '{print length($0);}'

# 例2.文字列「円周率は3」のバイト数を出力（文字列がUTF-8の場合、13が得られる）
echo '円周率は3'                      |
LC_ALL=C awk '{print length($0);}'
```

length 関数を使うにあたって注意しなければならないのは、環境変数で指定したロケールの通りに文字数を数えられるには、その環境にそのロケールがインストールされている必要がある。これは言い換えると、環境変数で設定されたロケールに従って、与えられたバイト列をそのロケールの文字列と認識できた場合にのみ文字数になる、ということである。従って、Shift-JIS 文字列のつもりで “LC_ALL=ja_JP.SJIS” などと設定しても、その環境が Shift-JIS ロケールに対応していなければバイト数で数えられるので注意が必要だ。

HACK 10.25　LEFT 関数

与えられた文字列の左端（先頭）から指定文字数の部分文字列を返す関数であるが、SUBSTR/SUBSTRING 関数の類で代用できるため、HACK 10.44 を参照。

HACK 10.26　LN / LOG / LOG10 関数

LN または（LN 関数が存在しない製品上の）LOG 関数は、自然対数を求める関数である。一方、（LOG10 関数が存在しない製品上の）LOG 関数または LOG10 関数は、常用対数を求める関数である。

SQL 文の例を示す。

■SQL

```
-- 例1．自然対数の底eの自然対数を出力（高精度であれば答えは1）
SELECT LN(2.71828182845905);

-- 例2．100の常用対数を出力（答えは2）
SELECT LOG(100);
```

UNIX の世界では AWK コマンドに log 関数があるが、これは自然対数を求める関数である。もし常用

対数を求める場合は、底の変換公式に基づき、

$$\log_{10} x = \frac{\log_e x}{\log_e 10} \tag{10.3}$$

として計算すればよい。

■UNIX

```
# 例1. 自然対数の底eの自然対数を出力（高精度であれば答えは1）
echo 2.71828182845905       |
awk 'BEGIN{OFMT="%.14f"; }  #
     {     print log($1);}'

# 例2. 100の常用対数を出力（高精度であれば答えは2）
echo 100                                    |
awk 'BEGIN{OFMT="%.14f";                  } #
     {     print log($1)/2.30258509299405;}'

# （例2について、常用対数を関数化した別解）
echo 100                                                |
awk 'BEGIN{OFMT="%.14f";                              } #
     {     print log10($1);                           } #
     function log10(x) {return log(x)/2.30258509299405;}'
```

AWK のデフォルト精度は小数点以下 6 桁だが、BEGIN セクションなどで “`OFMT="%.14f"`” と設定しておけば精度を 14 桁に上げられる。

HACK 10.27 LOWER 関数

LOWER（製品によっては LCASE）関数は、与えられた文字列を返す際、文字列内に大文字アルファベットが含まれていたらそれらをすべて小文字に置き換えてから出力する関数である。

SQL 文の例を示す。

■SQL

```
-- 例1. 住所録表"MY_ADDRESSES"
--       1列目. 番号列"number"
--       2列目. 住所1（ローマ字）列"address1"
--       3列目. 住所2（ローマ字）列"address2"
--       4列目. 住所3（ローマ字）列"address3"
--       5列目. 苗字（ローマ字）列"Myoji"
--       6列目. 名前（ローマ字）列"Namae"
--      を出力する際、住所1,2,3列はすべて小文字にする
SELECT "number"          AS "number"  ,
       LOWER("address1") AS "address1",
       LOWER("address2") AS "address2",
```

```
       LOWER("address3") AS "address3",
       "Myoji"           AS "Myoji"   ,
       "Namae"           AS "Namae"
FROM   "MY_ADDRESSES";

-- 例2. 例1のすべての列を小文字化して出力
SELECT "number"          AS "number"  ,
       LOWER("address1") AS "address1",
       LOWER("address2") AS "address2",
       LOWER("address3") AS "address3",
       LOWER("Myoji")    AS "Myoji"   ,
       LOWER("Namae")    AS "Namae"
FROM   "MY_ADDRESSES";
```

UNIX の世界に翻訳する場合は、AWK コマンドの tolower 関数を使うのが最も素直だ。ただし、テーブルファイル（データ）全体を一気に変換したい場合に限り tr コマンドを使うことができ、使える場合はこちらの方が高速である。

■UNIX

```
# 例1. 住所録表ファイル"MY_ADDRESSES.txt"
#       1列目. 番号列
#       2列目. 住所1（ローマ字）列
#       3列目. 住所2（ローマ字）列
#       4列目. 住所3（ローマ字）列
#       5列目. 苗字（ローマ字）列
#       6列目. 名前（ローマ字）列
#      を出力する際、住所1,2,3列はすべて小文字にする
awk '{print $1,tolower($2),tolower($3),tolower($4),$5,$6;}' "MY_ADDRESSES.txt"

# 例2. 例1のすべての列を小文字化して出力
cat "MY_ADDRESSES.txt" |
tr A-Z a-z
```

10.27.1　全角アルファベットも小文字化したい場合

SQL 文の LOWER/LCASE 関数は、一般的に全角の大文字アルファベットが与えられた場合にもきちんと小文字アルファベットを返してくれる。一方、AWK の tolower 関数はそれができない（ただし GNU AWK ではできる）し、tr コマンドでもできない。

しかし、Tukubai コマンドの zen/han コマンド（POSIX 版としても移植済）を使えばほぼ同等のことができる。

SQL 文の例は同じものとし、全角アルファベット対応版の UNIX 側のコードのみ示す。

■UNIX

```
# 例1. 住所録表ファイル"MY_ADDRESSES.txt"
#       1列目. 番号列
#       2列目. 住所1（ローマ字）列
#       3列目. 住所2（ローマ字）列
#       4列目. 住所3（ローマ字）列
#       5列目. 苗字（ローマ字）列
#       6列目. 名前（ローマ字）列
#      を出力する際、住所1,2,3列はすべて小文字にする
#
#     ※ コマンドの間にあるコメント行（"1:番号 2:住所1(半角化済)"など）は、
#        その位置を流れるデータの列構成を表している
#
cat "MY_ADDRESSES.txt"                                                  |
han 2 3 4                                                               |
# 1:番号 2:住所1(半角化済) 3:住所2(半角化済) 4:住所3(半角化済) 5:苗字 6:名前 #
awk '{print $1,tolower($2),tolower($3),tolower($4),$5,$6;}'             |
# 1:番号 2:住所1(半角化済) 3:住所2(半角化済) 4:住所3(半角化済) 5:苗字 6:名前 #
zen 2 3 4                                                               |
# 1:番号 2:住所1(全角化済) 3:住所2(全角化済) 4:住所3(全角化済) 5:苗字 6:名前

# 例2. 例1のすべての列を小文字化して出力
cat "MY_ADDRESSES.txt" |
han                    |
tr A-Z a-z             |
zen
```

zen コマンド[*5]と han コマンド[*6]は、引数で与えられた列番号の文字列中にある全角文字・半角文字を相互変換するものである。zen が「全角化」する方で、han が「半角化」する方だ。

アルファベットの大文字を小文字にするにあたっては、han コマンドを使って予め半角化する。こうすれば従来の方法が使える。大文字・小文字変換後に全角に戻すということを行っている。

もちろんこの方法だと、元々半角文字だったものまで最終的には全角文字に置き換わってしまう。従って、それでは困るというならこの方法は使えない。しかし、全角・半角を混ざった状態でないと困るというデータの方が珍しいのではないだろうか?

HACK 10.28 LTRIM 関数

→ TRIM 関数（HACK 10.48）参照

[*5] 詳細は次の URL を参照。
https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_zen

[*6] 詳細は次の URL を参照。
https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_han

HACK 10.29　MAX 関数

MAX 関数は、GROUP BY 句と共に用いる集約関数の一種で、集約される列の最大値を返す。
SQL 文の例を示す。

■SQL

```
-- 例．テストの成績表"EXAMSCORES"
--      1列目．学生ID列"id"
--      2列目．学期（1-3）列"term"
--      3列目．教科名列"subject"
--      4列目．点数列"score"
--     で、各学期・各教科の最高点を出力
--     列の構成は次の通り
--      1列目．学期（1-3）列"term"
--      2列目．教科名列"subject"
--      3列目．最高点"bestscore"
SELECT   "id"         AS "term"     ,
         "term"       AS "subject"  ,
         MAX("score") AS "bestscore"
FROM     "EXAMSCORES"
GROUP BY "term"   ,
         "subject";
```

UNIX の世界に翻訳するには、まずは処理を少しでも軽くするために、不要な列（学生 ID）列は削除する。そのうえで同じ学期、かつ同じ教科が隣り合うように sort コマンドを掛け、それを AWK コマンドに与えて同じ学期・教科が続く限り最大値かどうか比較して最後に残った値を出力する。

この方針で記述すると次のようになる。

■UNIX

```
# 例．テストの成績表ファイル"EXAMSCORES.txt"
#      1列目．学生ID列
#      2列目．学期（1-3）列
#      3列目．教科名列
#      4列目．点数列
#     で、各学期・各教科の最高点を出力
#     列の構成は次の通り
#      1列目．学期（1-3）列
#      2列目．教科名列
#      3列目．最高点
cat "EXAMSCORES.txt"                                              |
self 2 3 4                                                        |
# 1:学期 2:教科 3:点数                                           #
```

```
sort -b -k 2,2 -k 3,3                                                    |
awk 'BEGIN                  {# キー文字列,最高値を初期化                  #
                             if (getline) {prev_keys=$1 " " $2; max=$3;  #
                             } else       {prev_keys=""       ; exit  ;  } #
                                                                         #
     $1 " " $2 != prev_keys{print prev_keys,max;     # キーの最高値を出力  #
                             prev_keys=$1 " " $2;    # 新たなキーをセット  #
                             max      =$3;           # スコアをリセット  } #
                                                                         #
     {                       max = ($3>max)?$3:max;  # 最高値を更新      } #
                                                                         #
     END                    {# 未出力のキーの最高値を出力                 #
                             if (prev_keys=="") {exit;}                  #
                             print prev_keys,max;                       }'
```

HACK 10.30 MID / MIDB 関数

部分文字列を返す SUBSTR/SUBSTRING 関数の類と同じであるため、HACK 10.44 を参照。

HACK 10.31 MIN 関数

MAX 関数は、GROUP BY 句と共に用いる集約関数の一種で、集約される列の最小値を返す。SQL 文の例を示す。

■SQL

```
-- 例. テストの成績表"EXAMSCORES"
--      1列目. 学生ID列"id"
--      2列目. 学期 (1-3) 列"term"
--      3列目. 教科名列"subject"
--      4列目. 点数列"score"
--     で、各学期・各教科の最低点を出力
--     列の構成は次の通り
--      1列目. 学期 (1-3) 列"term"
--      2列目. 教科名列"subject"
--      3列目. 最低点"worstscore"
SELECT   "id"         AS "term"      ,
         "term"       AS "subject"   ,
         MAX("score") AS "worstscore"
FROM     "EXAMSCORES"
GROUP BY "term"   ,
         "subject";
```

UNIX の世界に翻訳するには、まずは処理を少しでも軽くするために、不要な列（学生 ID）列は削除する。そのうえで同じ学期、かつ同じ教科が隣り合うように sort コマンドを掛け、それを AWK コマンドに与えて同じ学期・教科が続く限り最小値かどうか比較して最後に残った値を出力する。

この方針で記述すると次のようになる。

■UNIX

```
# 例．テストの成績表ファイル"EXAMSCORES.txt"
#      1列目．学生ID列
#      2列目．学期（1-3）列
#      3列目．教科名列
#      4列目．点数列
#     で、各学期・各教科の最高点を出力
#     列の構成は次の通り
#      1列目．学期（1-3）列
#      2列目．教科名列
#      3列目．最低点
cat "EXAMSCORES.txt"                                                    |
self 2 3 4                                                              |
# 1:学期 2:教科 3:点数                                                  #
sort -b -k 2,2 -k 3,3                                                   |
awk 'BEGIN                 {# キー文字列,最低値を初期化                 #
                            if (getline) {prev_keys=$1 " " $2; min=$3;  #
                            } else       {prev_keys=""       ; exit  ;  } #
                                                                        #
    $1 " " $2 != prev_keys{print prev_keys,min;     # キーの最低値を出力  #
                           prev_keys=$1 " " $2;     # 新たなキーをセット  #
                           min      =$3;            # スコアをリセット  } #
                                                                        #
    {                      min = ($3<min)?$3:min;   # 最低値を更新      } #
                                                                        #
    END                    {# 未出力のキーの最低値を出力                #
                            if (prev_keys=="") {exit;}                  #
                            print prev_keys,min;                      }'
```

HACK 10.32　MOD 関数

MOD 関数は、割り算の余りを求める関数である。

SQL 文の例を示す。

■SQL

```
-- 例1．11割る7の余り出力（答えは4）
SELECT MOD(11,7);
```

```
-- 例2. -11割る7の余り出力（答えは-4）
SELECT MOD(-11,7);
```

UNIX の世界では、AWK コマンドの剰余の演算子 “%” を使えばよい。負数に対する剰余演算の考え方は複数あり、言語によって答えが別れる。しかし、被除数の絶対値に対して剰余演算を行った後で被除数の符号を与える方法が主流であり、SQL も AWK もこれを採用している。

■UNIX

```
# 例1. 11割る7の余り出力（答えは4）
# 例2. -11割る7の余り出力（答えは-4）
{ echo  11 7;
  echo -11 7 }          |
awk '{print $1 % $2;}'
```

HACK 10.33 NOW 関数

現在の年月日時分秒を返す関数であり、CURRENT_TIMESTAMP の類と同じであるため、HACK 10.13 を参照。

HACK 10.34 NVL 関数

→ COALESCE 関数（HACK 10.8）参照

HACK 10.35 PERIOD_DIFF 関数

→ 2 つの日時の差を求める関数であるため、「日時・時間の加減算はどうやるのか」（10.13.1 項）参照。

HACK 10.36 RAND / RANDOM 関数

RAND または RANDOM（製品によって名前が異なる）関数は、0 以上 1 未満の乱数を得る関数である。製品によっては第 1 引数に乱数のシード値を与えられるものもある。

SQL 文の例を示す。

■SQL

```
-- 例. 3行2列の乱数を出力
SELECT RAND(),RAND();
SELECT RAND(),RAND();
SELECT RAND(),RAND();
```

UNIX の世界に翻訳する場合、AWK コマンドにも乱数を発生させる関数 rand があることにはあるのだが、rand が内部的に利用している OS の乱数生成用のシステムコールが古くて低品質な場合がある。そこで、/dev/urandom という乱数源ファイルを利用することをお勧めする。そのために独自のユーザー関数

を用意した。ただしシードは与えられないので、それが必要な場合には AWK の組み込み関数である rand と srand を利用する。

■UNIX

```
# 例. 3行2列の乱数を出力（高品質な乱数）
awk 'BEGIN{
       OFMT="%.10f";                    # ←得られる桁数を増やす
       print random_num(),random_num();
       print random_num(),random_num();
       print random_num(),random_num();
     }
     function random_num( n) {
       "od -A n -t u4 /dev/urandom | \
        sed \"s/^ *//;s/  */ /g\"  | \
        tr \\  \\\\n"              | \
       getline n;
       return n/4294967296;

     }'

# 【別解】3行2列の乱数を出力（シード値を与える場合）
awk 'BEGIN{
       OFMT="%.10f";
       srand(1);              # ←ここにシード値
       print rand(),rand();
       print rand(),rand();
       print rand(),rand();
     }'
```

なお、最初の方法（random_num 関数）で作る乱数はもともと符号無し 32 ビット整数で、それを最大値 4294967296 で割ったものである。乱数が欲しいといっても、いつも 0 以上 1 未満の値が欲しいわけではなく、それを元に任意の範囲の整数値を作ることが主だと思う。そのような場合には、符号無し 32 ビット整数値を直接受け取るように関数を修正し、目的の範囲の整数を作るようにする方がよい。

HACK 10.37　REGEXP_REPLACE 関数

REGEXP_REPLACE 関数は、与えられた文字列を正規表現（拡張正規表現、ERE: Extended Regular Expression[*7]）に基づいて置換した結果を返す関数である。

書式については製品によって次のようなバリエーションがある。

- バリエーション A

[*7] 「正規表現メモ」というページの次の箇所が大変参考になる。
http://www.kt.rim.or.jp/ kbk/regex/regex.html#ERE

第 1 引数. 置換対象の文字列
第 2 引数. 正規表現パターン
第 3 引数. マッチした場合に置き換える文字列（省略した場合は空文字）
第 4 引数. フラグ（“i”:大文字小文字区別せず、“g”:グローバル置換）

- バリエーション B

第 1 引数. 置換対象の文字列
第 2 引数. 正規表現パターン
第 3 引数. マッチした場合に置き換える文字列（省略した場合は空文字）
第 4 引数. 置換開始位置（先頭文字を 1 とする、デフォルトは 1）
第 5 引数. マッチした場合の最大置換回数（0 なら無制限、デフォルトは 0）
第 6 引数. フラグ（“c”:大文字小文字区別、“i”:大文字小文字区別せず、“m”:複数行の行頭・行末にマッチ可能、“n”:ドットを改行文字にもマッチさせる、等）

SQL 文の例を示す。

■SQL

```
-- 例1. 文字列のうち最初の数値を"?"に置換（置換後は'a=?; b=45; c=6789'）
SELECT REGEXP_REPLACE('a=123; b=45; c=6789;','[0-9]+','?',1,1);

-- 例2. 文字列のうちすべての数値を消す（置換後は'a=; b=; c='）
SELECT REGEXP_REPLACE('a=123; b=45; c=6789;','[0-9]+','' ,1,0);
```

上記は、例 1 が 1 回のみの置換、例 2 がグローバル置換であり、UNIX の世界においては、これらはそれぞれ AWK コマンドの sub 関数、gsub 関数で行える。

■UNIX

```
# 例1. 文字列のうち最初の数値を"?"に置換（置換後は'a=?; b=45; c=6789'）
echo 'a=123; b=45; c=6789;'  |
awk '{str=$0;                #
      sub(/[0-9]+/,"?",s);   #
      print s;            }'

# 例2. 文字列のうちすべての数値を消す（置換後は'a=; b=; c='）
echo 'a=123; b=45; c=6789;'  |
awk '{str=$0;                #
      gsub(/[0-9]+/,"",s);   #
      print s;            }'
```

ただし、REGEXP_REPLACE 関数に比べていくつかの制約がある。

1 つは、AWK コマンドの正規表現では数量指定子（“{m,n}”、“{m,}”、“{,n}”、“{n}”）が使えないこと。もう 1 つは、グローバル置換以外の各種フラグや置換開始位置の指定に対応していないことだ。これらについては、AWK コマンドの他の関数を組み合わせるなどして対応するしかない。

HACK 10.38　REPLACE 関数

REPLACE 関数は、第 1 引数で与えられた文字列の中から、第 2 引数で与えられた文字列パターンを見つけ出し、第 3 引数で与えられた文字列に置換し、その結果を返す関数である。

UNIX の世界に翻訳する場合、最も近いのは AWK コマンドの sub 関数である。しかしこれは正規表現メタ文字を認識してしまうので、翻訳する際にはそれらが認識されてしまわないように、それらメタ文字を 1 つ 1 つエスケープ（メタ文字の手前に “\” を置くことを）しなければならない。

このことを前提にして、続きは REGEXP_REPLACE 関数（HACK 10.38）を参照されたい。

HACK 10.39　RIGHT 関数

与えられた文字列の右端（末尾）から指定文字数の部分文字列を返す関数であるが、SUBSTR/SUBSTRING 関数の類で代用できるため、HACK 10.44 を参照。

HACK 10.40　ROUND 関数

ROUND 関数は、与えられた数値の四捨五入を行う関数である。製品によっては、第 2 引数 n を指定して小数点第 n 位に四捨五入する（1 を指定した場合、1.09 が 1.1 になる）こともできる。

SQL 文の例を示す。

■SQL

```
-- 例1. 1.58の四捨五入値（整数）を出力（答えは2）
SELECT ROUND(1.58);

-- 例2. -1.58の四捨五入値（小数第1位まで）を出力（答えは-1.6）
SELECT ROUND(-1.58,1);
```

UNIX の世界では、AWK コマンドの int 関数を使って（端数があれば）0.5 足した後に四捨五入……と言いたいところだが、負の値四捨五入のルールでは「絶対値として四捨五入した後で負号を付ける」というルールがあるため、単純にはいかない。そこで AWK のユーザー関数として同名関数を用意した。あとは必要な時にこのコード（スニペット）を流用すればよい。

■UNIX

```
# 例1. 1.58の四捨五入値（整数）を出力（答えは2）
echo 1.58 |
awk '{print round($1);}
     function round(f,n) {
       n=n*1;
       f=f*10^n;
       if (int(f)==f) {return       f      /10^n;}
       if (f > 0    ) {return  (int(f+0.5))/10^n;}
```

```
                    return -(int(0.5-f))/10^n;
    }'

# 例2. -1.58の四捨五入値（小数第1位まで）を出力（答えは-1.6）
echo -1.58 1 |
awk '{print round($1,$2);}
    function round(f,n) {
      n=n*1;
      f=f*10^n;
      if (int(f)==f) {return       f      /10^n;}
      if (f > 0    ) {return  (int(f+0.5))/10^n;}
                    return -(int(0.5-f))/10^n;
    }'
```

HACK 10.41 RTRIM 関数

→ TRIM 関数（HACK 10.48）参照

HACK 10.42 SIN 関数

COS 関数は、正弦（サイン）値を求める関数である。引数の単位はラジアン。
SQL 文の例を示す。

■SQL

```
-- 例. sin(pi)の値を出力（高精度であれば答えは0）
SELECT SIN(3.14159265358979);
```

UNIX の世界では、AWK コマンドの cos 関数がこれに相当するので、それを使えばいい。

■UNIX

```
# 例. sin(pi)の値を出力（高精度であれば答えは0）
echo 3.14159265358979       |
awk 'BEGIN{OFMT="%.14f"; }  #
    {     print sin($1);}'
```

AWK のデフォルト精度は小数点以下 6 桁だが、BEGIN セクションなどで “`OFMT="%.14f"`” と設定しておけば精度を 14 桁に上げられる。

HACK 10.43 SQR / SQRT 関数

SQR または SQRT 関数（製品によって名前が異なる）は、正の平方根（ルート）を求める関数である。
SQL 文の例を示す。

■SQL

```
-- 例．ルート2の値を出力（高精度であれば答えは約1.41421356237310）
SELECT SQRT(2);
```

UNIX の世界では AWK コマンドの sqrt 関数がこれに相当するので、それを使えばいい。

■UNIX

```
# 例．ルート2の値を出力（高精度であれば答えは約1.41421356237310）
echo 2                          |
awk 'BEGIN{OFMT="%.14f";  }  #
     {     print sqrt($1);}'
```

AWK のデフォルト精度は小数点以下 6 桁だが、BEGIN セクションなどで “`OFMT="%.14f"`” と設定しておけば精度を 14 桁に上げられる。

HACK 10.44　SUBSTR / SUBSTRING / SUBSTRB 関数

SUBSTR や SUBSTRING 関数は、引数で与えられた文字列のうち、指定位置から指定文字数の部分文字列を返す関数である。SUBSTRB は、指定位置や指定文字数をバイト単位で指定することを明示したものである。（製品によって使える関数の名前や書式が違う）

ここでは PostgreSQL の SUBSTRING 関数を例にする。

■SQL

```
-- 例．文字列「円の円周率は3.14」から「円周率は3」の部分を出力
SELECT SUBSTRING('円の円周率は3.14' from 3 for 5);
```

UNIX の世界に翻訳するなら、（sed コマンドで実装する方法もあるが）最も素直に翻訳できるのは AWK コマンドの substr 関数である。ただし、位置や長さを文字列数で数えるかバイト数で数えるかは環境変数（LANG や LC_CTYPE、あるいは LC_ALL）の設定によって変わる。

■UNIX

```
# 例．文字列「円の円周率は3.14」から「円周率は3」の部分を出力
echo '円の円周率は3.14'                              |
LC_ALL=ja_JP.UTF-8 awk '{print substr($0,3,5);}'

#  （文字列はUTF-8だったとし、同じことをバイト数で行う場合）
echo '円の円周率は3.14'                    |
LC_ALL=C awk '{print substr($0,7,13);}'
```

substr 関数を使うにあたっての注意点は 3 つある。

1 つは、第 2 引数の開始位置の最小値は 1 である。つまり先頭の文字は 0 ではなく 1 であるということ。これは SQL の多くの同等関数と同じである。

次に、第 3 引数に負値は設定**できない**ということ。SQL 文で負値を使っていてそれを翻訳する場合には、AWK コマンドに翻訳する際、length 関数を使って文字列長を求めた上で正の数としての位置を計算してから与えること。

最後に、環境変数で指定したロケールの通りに文字数を数えられるには、その環境にそのロケールがインストールされている必要がある。言い換えれると、環境変数で設定されたロケールに従って、与えられたバイト列をそのロケールの文字列と認識できた場合にのみ文字位置・文字数として解釈される、ということである。従って、Shift-JIS 文字列のつもりで “LC_ALL=ja_JP.SJIS” などと設定しても、その環境が Shift-JIS ロケールに対応していなければバイト数で数えられるので注意が必要だ。

HACK 10.45 SUM 関数

SUM 関数は、GROUP BY 句と共に用いる集約関数の一種で、集約される列の合計値を返す。
SQL 文の例を示す。

■SQL

```
-- 例．テストの成績表"EXAMSCORES"
--      1列目．学生ID列"id"
--      2列目．学期（1-3）列"term"
--      3列目．教科名列"subject"
--      4列目．点数列"score"
--     で、各学生の1つの学期の総合点を出力
--     列の構成は次の通り
--      1列目．会員ID列"id"
--      2列目．学期列"term"
--      3列目．総合得点"totalscore"
SELECT   "id"          AS "id"         ,
         "term"        AS "term"       ,
         SUM("score") AS "totalscore"
FROM     "EXAMSCORES"
GROUP BY "id"   ,
         "term";
```

UNIX の世界に翻訳するには、まずは同じ学生 ID、かつ同じ学期が隣り合うように sort コマンドを掛け、その後 AWK コマンドを用いて同じ学生 ID かつ同じ学期が続く限り、点数を足して合計値を求めてもよいが、Tukubai コマンドセットにある同様の動作をする sm2 というコマンドを用いて簡潔に記述することを勧める。

■UNIX

```
# 例．テストの成績表ファイル"EXAMSCORES.txt"
#      1列目．学生ID列
#      2列目．学期（1-3）列
#      3列目．教科名列
#      4列目．点数列
#     で、各学生の1つの学期の総合点を出力
#     列の構成は次の通り
#      1列目．会員ID列
```

```
#       2列目．学期列
#       3列目．総合得点
cat "EXAMSCORES.txt"  |
sort -b -k 1,1 -k 2,2 |
sm2 1 2 4 4
```

sm2 コマンドの詳細は次の Web ページ[*8]に載っているが、ここに示した例の意味は 1 列目〜2 列目が同じ値である行が続く限り、4 列目〜4 列目（つまり 4 列目のみ）を足して得られた合計値を返せという意味だ。一般化して説明すると、“sm2　a b c d” と記述された場合、「a 列目から b 列目までの値がすべて同一の行の、c 列目から d 列目の値をそれぞれ（c 列目から d 列目を足し合わせるのではない）の合計値を求める」という動作をする。注意点としては、例でも示したように、同一値であることを見るすべての列に対して事前ソートを行っておくこと。

なお、この sm2 は Tukubai コマンドセットにおける sm（sum-up）シリーズの一種であり、他に sm4、sm5 というコマンドがある。詳細はそれぞれの man ページ[*9][*10]を参照してもらいたいが、ここでは参考までに、sm4 の使用例を示す。

■UNIX

```
# 例．テストの成績表ファイル"EXAMSCORES.txt"
#       1列目．学生ID列
#       2列目．学期（1-3）列
#       3列目．教科名列
#       4列目．点数列
#      で、各学生の1つの学期の総合点を出力したいが、
#      その各教科の点数を一通り列挙した後で、
#      合計点の行を追加して出力せよ
#      よって列構成は元の通りとする
cat "EXAMSCORES.txt"          |
sort -b -k 1,1 -k 2,2 -k 3.3 |
sm4 1 2 3 3 4 4
```

“sm2　$a1$ $a2$ $b1$ $b2$ $c1$ $c2$” と記述された場合、$a1$〜$a2$ 列がすべて同一の列（キー列）について、$b1$〜$b2$ 列をサブキー列と見なし（=そのまま出力し）、$c1$〜$c2$ 列の値をそれぞれ（$c1$〜$c2$ 列を横に合計するのではなく）の合計値の行を追加するという動作をする。

したがって、

```
G2_01 1 英語 92
G2_01 1 国語 68
G2_01 1 数学 30
G2_02 2 英語 95
G2_01 2 国語 88
G2_01 2 数学 54
```

[*8] https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_sm2
[*9] https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_sm4
[*10] https://uec.usp-lab.com/TUKUBAI_MAN/CGI/TUKUBAI_MAN.CGI?POMPA=MAN1_sm5

```
G2_02 3 英語 98
G2_01 3 国語 81
G2_01 3 数学 60
  :   :  :    :
```

というデータがあって、これを “sm4 1 2 3 3 4 4” に通すと、

```
G2_01 1 英語 92
G2_01 1 国語 68
G2_01 1 数学 30
G2_01 1 @ 190
G2_02 2 英語 95
G2_01 2 国語 88
G2_01 2 数学 54
G2_01 2 @ 237
G2_02 3 英語 98
G2_01 3 国語 81
G2_01 3 数学 60
G2_01 3 @ 239
  :   :  :    :
```

という結果が得られる（合計値行のサブキー文字列のデフォルトは “@” とされる）。集約前の行を残しつつ、合計値行を求めるというのは SQL 文では書きづらいように思う。

HACK 10.46 TAN 関数

TAN 関数は、正接（タンジェント）値を求める関数である。引数の単位はラジアン。

SQL 文の例を示す。

■SQL

```
-- 例. tan(pi)の値を出力（高精度であれば答えは0）
SELECT SIN(3.14159265358979);
```

UNIX の世界では AWK コマンドを用いるのが最も簡単ではあるものの、tan 関数に相当するものが無い。しかし、tan は

$$\tan x = \frac{\sin x}{\cos x} \tag{10.4}$$

という式で求められるので、AWK の sin 関数と cos 関数の組み合わせで得られる。

■UNIX

```
# 例. tan(pi)の値を出力（高精度であれば答えは0）
echo 3.14159265358979               |
awk 'BEGIN{OFMT="%.14f";         }  #
     {     print sin($1)/cos($1);}'
```

AWK のデフォルト精度は小数点以下 6 桁だが、BEGIN セクションなどで “OFMT="%.14f"” と設定し

ておけば精度を 14 桁に上げられる。

HACK 10.47　TIMEDIFF / TIMESTAMPDIFF 関数

→ 2 つの日時の差を求める関数であるため、「日時・時間の加減算はどうやるのか」（10.13.1 項）参照。

HACK 10.48　TRIM 関数

TRIM 関数は、与えられた文字列から、その先頭・末尾にある空白の連続を除去したものを返す関数である。ここでは、先頭（左側）のみ行う LTRIM 関数、末尾（右側）のみ行う RTRIM 関数も合わせて説明する。

■SQL

```
-- 例．会員表"MEMBERS"
--      1列目．会員ID列"id"
--      2列目．苗字列"Myoji"
--      3列目．名前列"Namae"
--         :
--     の、1〜3列目を出力
--     ただし、苗字列・名前列の先頭・末尾に空白があればすべて取り除くこと
SELECT   "id"          AS "id"   ,
         TRIM("Myoji") AS "Myoji",
         TRIM("Namae") AS "Namae"
FROM     "MEMBERS";
```

UNIX の世界に翻訳するなら、AWK の正規表現置換関数 gsub を使って取り除くのが簡単だ。ただし、HACK 2.2 で取り決めたルールによって、前後の空白はアンダースコア “_” にしているはずなのでそれを取り除く。

■UNIX

```
# 例．会員表ファイル"MEMBERS.txt"
#      1列目．会員ID列
#      2列目．苗字列
#      3列目．名前列
#         :
#     の、1〜3列目を出力
#     ただし、苗字列・名前列の先頭・末尾に空白（アンダースコア"_"）があれば
#     すべて取り除くこと
cat "MEMBERS.txt"                          |
awk '{myoji=$2; gsub(/^_*|_*$/,"",myoji);   #
      namae=$3; gsub(/^_*|_*$/,"",namae);   #
      print $1, myoji, namae;            }'
```

gsub とは、第 3 引数の文字列変数に対し、第 1 引数の正規表現パターンにマッチする部分文字列を、第

2 引数の文字列に置換（グローバル置換、該当する文字列パターンがあれば何度でも）するという AWK の関数である。これを応用すれば、LTRIM 相当も RTRIM 相当もできる。

TRIM 相当 gsub(/^_*|_*$/,"", 変数名)
LTRIM 相当 sub(/^_*/,"", 変数名)
RTRIM 相当 sub(/_*$/,"", 変数名)
※ LTRIM、RTRIM 相当ではグローバル置換の必要がないため、sub 関数を用いている。

10.48.1 全角空白もあわせてトリミングしたい場合

SQL 文の TRIM、LTRIM、RTRIM 関数はどれも、半角空白のみならず全角空白も文字列の前後にあればトリミング対象としてくれる。では UNIX の世界ではどうするかといえば、正規表現のパターンを微修正するだけである。

SQL の例は同じとして、UNIX 側の例のみ示す。

■UNIX – 全角空白も考慮する場合

```
# 例．会員表ファイル"MEMBERS.txt"
#       1列目．会員ID列
#       2列目．苗字列
#       3列目．名前列
#          :
#      の、1〜3列目を出力
#      ただし、苗字列・名前列の先頭・末尾に全角含む空白（アンダースコアや全角空白）
#      があればすべて取り除くこと
cat "MEMBERS.txt"                                                    |
LC_ALL=ja_JP.UTF-8 awk '{myoji=$2; gsub(/^[_　]*|[_　]*$/,"",myoji);   #
                         namae=$3; gsub(/^[_　]*|[_　]*$/,"",namae);   #
                         print $1, myoji, namae;                    }'
```

“[_　]” の部分が分かりにくいが、アンダースコアの直後に記してあるのは全角空白である。

HACK 10.49 TO_*関数（データ型変換）

→ CAST 関数（HACK 10.6）参照

HACK 10.50 UCASE 関数

→ UPPER 関数（HACK 10.51）参照

HACK 10.51　UPPER 関数

UPPER（製品によっては UCASE）関数は、与えられた文字列を返す際、文字列内に小文字アルファベットが含まれていたらそれらをすべて大文字に置き換えてから出力する関数である。

SQL 文の例を示す。

■SQL

```
-- 例1．住所録表"MY_ADDRESSES"
--       1列目．番号列"number"
--       2列目．住所1（ローマ字）列"address1"
--       3列目．住所2（ローマ字）列"address2"
--       4列目．住所3（ローマ字）列"address3"
--       5列目．苗字（ローマ字）列"Myoji"
--       6列目．名前（ローマ字）列"Namae"
--      を出力する際、住所1,2,3列はすべて大文字にする
SELECT "number"          AS "number"  ,
       UPPER("address1") AS "address1",
       UPPER("address2") AS "address2",
       UPPER("address3") AS "address3",
       "Myoji"           AS "Myoji"   ,
       "Namae"           AS "Namae"
FROM   "MY_ADDRESSES";

-- 例2．例1のすべての列を小文字化して出力
SELECT "number"          AS "number"  ,
       UPPER("address1") AS "address1",
       UPPER("address2") AS "address2",
       UPPER("address3") AS "address3",
       UPPER("Myoji")    AS "Myoji"   ,
       UPPER("Namae")    AS "Namae"
FROM   "MY_ADDRESSES";
```

UNIX の世界に翻訳する場合は、AWK コマンドの toupper 関数を使うのが最も素直だ。ただし、テーブルファイル（データ）全体を一気に変換したい場合に限り tr コマンドを使うことができ、使える場合はこちらの方が高速である。

■UNIX

```
# 例1．住所録表ファイル"MY_ADDRESSES.txt"
#       1列目．番号列
#       2列目．住所1（ローマ字）列
#       3列目．住所2（ローマ字）列
#       4列目．住所3（ローマ字）列
```

```
#       5列目．苗字（ローマ字）列
#       6列目．名前（ローマ字）列
#      を出力する際、住所1,2,3列はすべて小文字にする
awk '{print $1,toupper($2),toupper($3),toupper($4),$5,$6;}' "MY_ADDRESSES.txt"

# 例2．例1のすべての列を小文字化して出力
cat "MY_ADDRESSES.txt" |
tr a-z A-Z
```

10.51.1 全角アルファベットも大文字化したい場合

この内容については 10.27.1 を参照。

あとがき

● 著者コメント

リッチー大佐の中の人
POSIX 原理主義で世界制覇を目論む組織「秘密結社シェルショッカー」の大幹部。資金獲得のため、戦闘力を上げるため、また我々の思想で人間たちを染め上げるため、世を忍ぶ仮の姿で、業務システム構築を請け負ったり、講師をしたり、執筆をしたりしている。この本の内容も、そうした実戦経験を積んで会得してきた戦闘法が元になっているのだ。

もちろんどの案件も POSIX 原理主義シェルスクリプトで、である。シェルスクリプト以外に言語が縛られる案件など引き受けたら粛清ものだ。組織の厳しい掟を背負いながら、日々どこかで暗躍している。

この本も 1.0 になり、POSIX 原理主義者を世界制覇を成し遂げるという野望にまた一歩近づいた。さぁ、お前たちもこれを読んで POSIX 原理主義に改宗するのだ！そしてもし、寿命の短いシステム開発、あるいはソフトウェア製品に苦しんでいるのなら、下記のメールアドレスに相談しにきてもいいぞ。
E-mail：`richie.shellshoccar@gmail.com`

● 表紙担当者コメント

もじゃ
表紙担当のもじゃである。どこにでもいる SE で、普段はもっぱらもじゃもじゃしている。

（2018 年のはなし）
今年の夏はフィアンセ（架空の人物・実在はしない）と海ほたるにドライブに行く約束をしている。実に楽しみである。帰りに彼女の飼っている猫（架空の生き物・実在はしない）がカワイイらしいので会いに行けたらいいなと思っている。

File/Dir Hacks — POSIX 原理主義者が教える最強データ管理術

2018 年 8 月 10 日	0.1 版（基礎編として）発行
2019 年 12 月 31 日	0.2 版（基礎 + トランザクション編として）発行
2020 年 6 月 1 日	0.21 版（誤字修正版）発行
2021 年 12 月 31 日	1.0 版（SQL マイグレーション編追記版）発行
2022 年 8 月 27 日	1.01 版（1.0 版の誤り訂正版）発行

著　　者	リッチー大佐
表　　紙	もじゃ
制 作 協 力	321516（三井浩一郎）
印刷・製本	Comflex
発行・発売	松浦リッチ研究所
	https://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

FILE/DIR HACKS

File/Dir Hacks — SQL教徒をPOSIX原理主義者にする最強データ管理術

2016年、Windows Subsystem for Linuxの登場により、遂にWin/Mac/UNIXの世界制覇を果たし、最強の開発手法になったPOSIX原理主義。これを採用すれば、最低限のUNIX環境だけで開発ができ、もう何も他の言語やライブラリー等をインストールする必要などない。もちろんRDB製品もである。しかしUNIX環境を与えられ、さあデータ管理しろと言われただけでは難しい。そこで本書では、ファイルやディレクトリーを駆使した効果的なデータ格納法、トランザクション処理のテクニック、そしてSQLの様々な句と同等の処理をUNIX・シェルスクリプトで記述する方法まで解説する。

Rich Lab.　発行所／まつらリッチ研究所

シェルショッカー　影の発行元／秘密結社シェルショッカー日本支部
イベント頒価 1000円