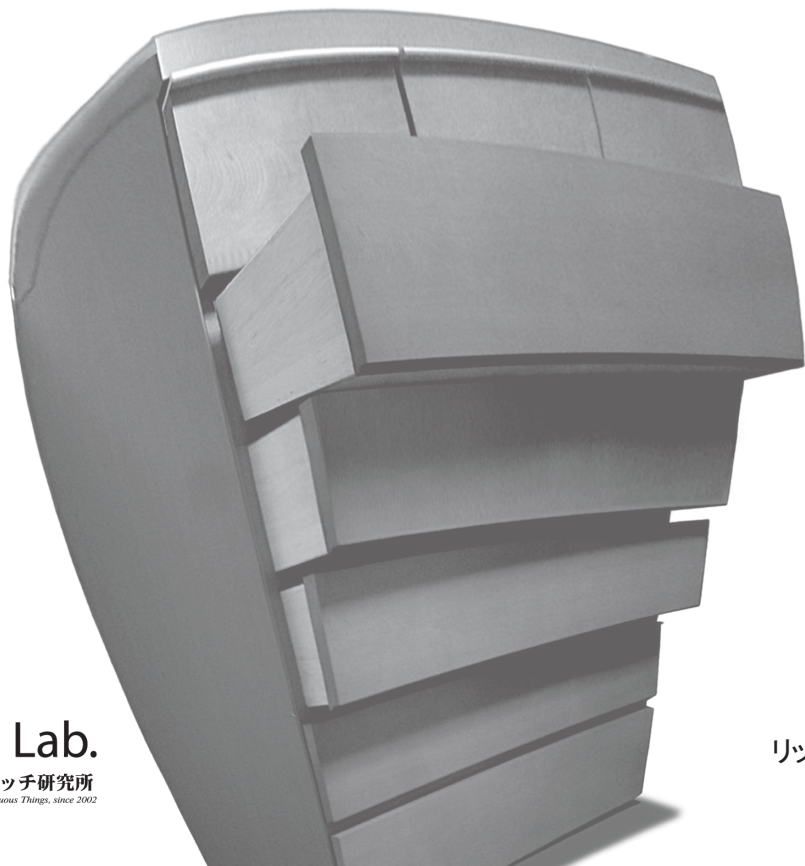


FILE/DIR HACKS

ver.0.2
基礎+
トランザクション編

POSIX原理主義者が教える
最強データ管理術



File/Dir Hacks

POSIX 原理主義者が教える最強データ管理術

リッチー大佐 著



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。

実的な例と言われれば、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.0.2（基礎＋トランザクション編）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たが、そこで終わってしまった。次の版では完成させようとしていたが、他の活動に阻まれてなかなか完成せず、再び仮リリースすることになった。

この0.2版では、データ管理という本題から少々脇道にそれて、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という話題を、4章～6章として追加した。「シェルスクリプトではトランザクション処理できないでしょ」というツッコミを以前からもらっていたのでその回答を早く書きたいという理由もあった。また、0.1版の範囲（1～3章）も若干加筆・修正してある。

本書の完成は……、来年5月の夏コミにこそは達成したい。その時こそ、世の中の典型的なデータ加工の数々（SQLにあるような数々の加工）が、UNIXコマンドやシェルスクリプトの流儀でどう鮮やかにこなせるかを存分に堪能できることだろう。

そしてまだ未完成な状態であるため、表紙は「例の本」っぽいのに、本文のレイアウトはまだ \LaTeX そのままなのはご愛敬。もう少し内容が充実してきたら、本文も真面目にパクるオマージュするので改訂版に期待を。（技術書典にも出展しているオ○イリーさんなら怒らないよね）

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。

<http://richlab.org/pub/ebook/fdh0.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	7
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
第 3 章	データ管理のためのディレクトリーの使い方	31
HACK 3.1	RDB のテーブル 1 つ ≠ ファイル 1 つ	31
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	32
3.1.2	ファイルなら、階層化して格納できる	32
HACK 3.2	1 つのディレクトリー直下のファイル数 ≤ 10000	33
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	33
3.2.2	考察	34
HACK 3.3	内容更新する 1 つのファイルに収める行数 ≤ 10000	34
3.3.1	大きい 1 つのファイルと細かい大量のファイル	34
3.3.2	1000 万レコードファイルの持ち方実験	35
3.3.3	1 ファイルに持たせるべきレコード数の上限は?	37
3.3.4	会員データはどう管理するか	37
第 4 章	実は排他制御不要な排他処理	39
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	39
4.1.1	ファイルサイズをカウンターとして使う	39
4.1.2	なぜファイルロック不要なのか	40
4.1.3	デメリットと限界	41
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	42
HACK 4.3	ファイル更新の最後には mv コマンドを使う	42
4.3.1	良い例と悪い例	43
HACK 4.4	小さなファイルをディレクトリーに並べる	43
4.4.1	在庫数ファイルによる例	44

4.4.2	有効性が低いケースとその対策	44
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	45
4.5.1	ウォーターフォール書き出しが好ましい理由	46
4.5.2	保管コストとの折り合い	46
HACK 4.6	その他、アトミックな処理を活用する	47
4.6.1	アトミックと見なせるファイル操作	47
第 5 章	排他処理（トランザクション処理）	49
HACK 5.1	排他ロックは「ln などの早い者勝ち」ルールで捌く	49
5.1.1	2 種類のロック	49
5.1.2	排他ロックの考え方	50
5.1.3	排他ロック実装のための具体的なルール	51
5.1.4	排他ロックコマンド “pexlock”	51
5.1.5	制約事項	54
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	54
5.2.1	共有ロックの考え方	55
5.2.2	共有ロック管理のためのデータ構造はどうすべきか	55
5.2.3	共有ロック管理のための各種ファイル操作	57
5.2.4	共有ロックコマンド “pshlock”	59
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	62
5.3.1	セマフォとは	62
5.3.2	ファイル・ディレクトリーを駆使して実現する	62
5.3.3	リソーススターベーション問題	64
第 6 章	非同期キューイング - 食券方式な食堂の注文管理	65
HACK 6.1	キューイングは、食券方式の食堂を真似する	65
6.1.1	飲食店の食券はどういう仕組みか	65
6.1.2	食券方式をファイル・ディレクトリーで再現する	67
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	71
6.2.1	大規模な食堂はどうなっているか	71
6.2.2	ファイル・ディレクトリーで再現する	72
あとがき		73

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」。そんな数々の苦勞を強いられながら RDB 製品を使っているというなら、もうそんな苦勞とはおさらば。さあ、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ただけの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、これから話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

```
17B243051100020049004
```

*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ただきたい。

■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つ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。

各論については、改訂版でおいおい述べていきたい。

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か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更ってしまった
 - 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しているとは言えない。

では、強韌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、高度なビューア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していれば、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。

と思いはしなかつたらどうか。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。

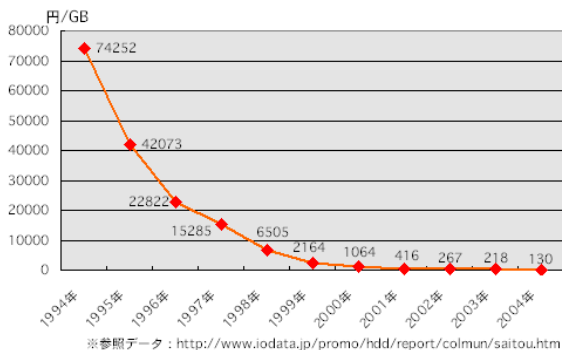


図 1.1 HDD 容量単価の推移 出典:http://pc.nf4hou.com/hardware/hdd_kakaku.html

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** しておくべきだ。

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。そういう時でも、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つてのことだ。

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どということだ。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。

したがって、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。

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。

^{*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。



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

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

^{*3} <https://uec.usp-lab.com/tukubai>

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。とはいつ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せ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。

準備ができれば、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/shellshoccar.sh↵  
$
```

(curl コマンドがある場合)

```
$ curl -o https://raw.githubusercontent.com/ShellShoccar-jpn/installer/master/shellshoccar.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、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。

同じ道を進ってきた我々が、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ひします。"
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行った。

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】では列区切り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ぬ。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 (ビーブ音)	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、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、あま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ない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 하나가 US (0x1F) というわけだ。

どうやって使うかという、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もので、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ない、「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ま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/_/\_\/'          | # ←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 部もともと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みであり、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、“unesj.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ひします。
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ひします。"
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 }' > grpmsst.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 -i 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 -i 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。

^{*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。

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 \neq 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ると言われたら、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の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・削除が自由に行える*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、その代わりに、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のが難しかったのに対し、ファイルはディレ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だければ、レコードの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、こ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、大量の細かい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、ShellSoccar コマンドセットに収録されている“greo”（Global Regular Expression and **O**verwrite）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、その名のとおりに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 のオーダーでもいざしろし、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らろうし、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らろう。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 文字目も見ればよい。

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います。

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 Tukupai から移植した“up3”というコマンド*6があった。さらに使いやすいのだが、その解説は本書の今後の改訂版で追加する予定である。

*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かくのではなく、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
```

などとやればよい。

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で割り込まない）。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。更はその 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ふので、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なり、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、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さ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一番事故に遭いやすい。悪い例 2 では一時ファイルを經由して 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）終わらせる。

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@[^:]*@/00' |
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方がいいが、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、二度とやり直せない、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。

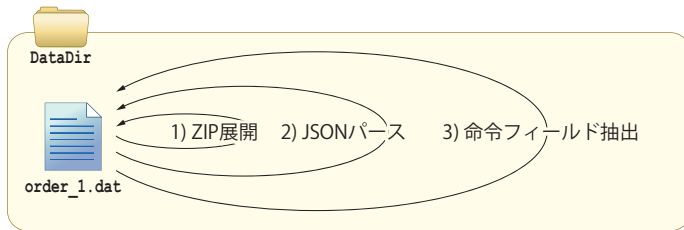


図 4.1 【a】 データ加工をする都度、同じファイルに上書きして更新していく

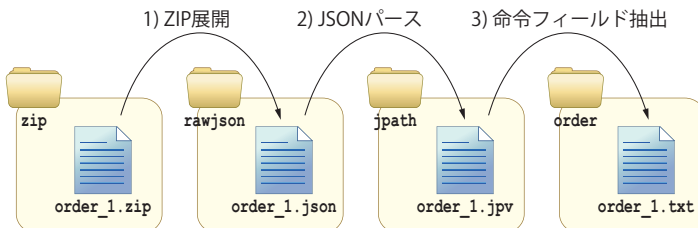


図 4.2 【b】 データ加工をする都度、次々と別のファイルに書き出していく

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素所で残してあれば、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」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「In 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種類ある。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。

- 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`
- 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`
- 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`
- 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えば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、この名前で作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。したがって、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³。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。

したがって、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/                        ← 同じ名前で作るサブdir
|       |
|       +-- <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り、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…
$ ls sh_lock1 ↵             ← 改名されず既存 dir の中に入ってしまった!
.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
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ー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、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
[ $n -gt 0 ] && {
  rm "$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、残っているの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、指定された名前で作成された共有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、この名前でも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。したがって、この場所が違えば同じ名前でも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に指定も可)。
 - 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を探す。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
- `pclock` - 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/$name/.busy") || 1) 排他ロックを試み、
continue                                         失敗したら最初へ戻り、リトライ
n=(ls -lf "$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)
* * * * * pclllock -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ー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、本章で見え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ず店。(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。

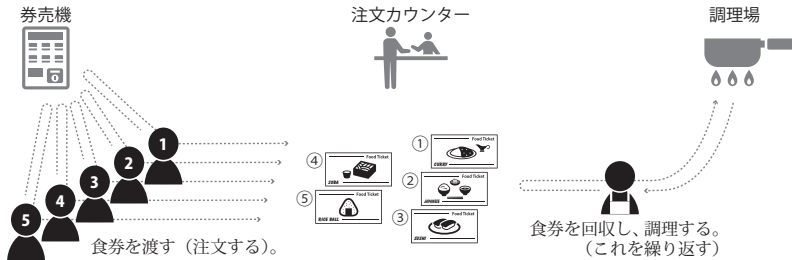


図 6.1 食券方式の食堂（小規模な場合）

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。
- 配膳まで終わっ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。もし複数箇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するために、非同期式の利点を達成するのに、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。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ます。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はないが、混雑度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すべての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 で既に説明済みだ。

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な。

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すべてのプログラム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なくては困るからだ。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增強して、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いる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。

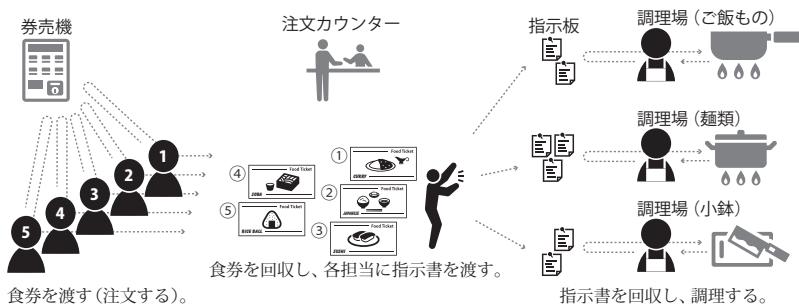


図 6.2 食券方式の食堂（大規模な場合）

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。

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。

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肅清ものだ。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。

本書を完成させ、POSIX 原理主義者を世界中に増やすという野望があるが、今回は間に合わなかった。おのれえ! 今度の5月の夏コミにこそは完成させてやる。それまで待っているがいい。

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、普段はもっばらもじゃもじゃしている。

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ど思っている。

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

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
<http://richlab.org/>
影の発行元 秘密結社シェルショッカー日本支部

Printed in Japan

FILE/DIR HACKS

File/Dir Hacks — POSIX原理主義者が教える最強データ管理術

2016年の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製品でもある。しかしUNIX環境だけ与えられ、さあこれでデータ管理をしろと突然言われたら、戸惑うのも無理はない。本書では、ファイルやディレクトリー、そしてUNIXコマンドを駆使することで実現する、データ管理術のほか、トランザクション処理などのテクニックまでも解説する。

Rich Lab.

発行所／まつらリッチ研究所



影の発行元／秘密結社シエルシヨッカー—日本支部
頒布価格 500円
