インターフェイスの街角— タイムスタンプの有効活用

増井 俊之

ファイル名の憂鬱

UNIX や Windows などのひろく使われている計算機システムでは、永続的なデータはすべてファイルとして管理するようになっています。したがって、どのような性質のデータであっても、なんらかの名前を付けて適当なファイルに格納する必要があります。

コマンド間のパイプを通過するデータのような一時的に使うデータは名前を付けなくてもよいのに、ディスク上に格納するあらゆるデータにかならず適当な名前を付けなければならないのは、けっこう面倒なものです。

たとえば、以下のような種類の情報については、保存しておく必要はあっても、いちいち名前を付けるのはなんとなく億劫です。

- メモや雑記
- 手書きメモ
- 支払いメモ
- 記録写真
- 日記
- 作業や実験のログ
- インストールメモ
- 開発メモ
- ダウンロードしたファイル（いろいろなファイルをダウンロードしても、どれがどのようなファイルだったか、すぐに忘れてしまうからです）

これらのようなファイルは、データとしては重要かもしれませんが、どこにどのような名前で格納すべきかなどと、けっこう悩むものです。私の場合、とくにインストールメモや試行錯誤を繰り返すような実験のログにいちいち名前を付けるのはとくに面倒に感じますし、せっかく名前を付けて保存しても、いつの間にか忘れてしまったりします。また、ダウンロードした記録などは、まったくうまく整理できず困っています。

一般に、各種のメモのような雑多なデータは、計算機では PIM (Personal Information Manager) で扱うことが多いようです。Palm などの PDA では個々の情報にいちいち名前を付ける必要はなく、ファイル名で悩むことはありません。Palm の場合、メモ帳の個々のデータには名前が付いておらず、先頭 1 行目の内容一覧や全文検索機能で必要なものを捜すようになっています。この方式でも実用上はとくに問題はありませんし、ファイル名を考えなくてよいぶん、気分的に楽になります。

Windows の世界では、タイトルやファイル名を気にすることなく、いろいろな情報をまとめて整理できる「紙 2001」などのソフトに人気があるようです。UNIX システム上で PDA と同様に気軽にデータを扱えるようにするには、永続的なデータに名前を付けずに管理する方法が必要でしょう。

タイムスタンプと一時ファイル

名前に頭を悩ますことなくデータをファイルとして格納したい場合、データの作成/更新時刻をファイル名に使うと便利です。

複数の情報に同じ時刻が記録されていれば、その時刻にもとづいてデータを関連づけることができます。たとえば、データが作成されたりダウンロードされたりした時刻をメモに記録しておけば、メモからそれらのデータにアク

図 1　講演のメモ

セスすることは容易でしょう。

たとえば、以下に述べるような場面では、時刻を活用すれば情報をうまく連繋させることができるはずです。

写真とメモによる記録

講習会や展示会などでは、写真を撮ったりメモをとったりを繰り返すことがよくあります。デジタルカメラで撮った写真は画像データ用のディレクトリに格納されることが多く、メモなどの文書とは異なる場所に置かれるのが普通です。次のようにしてメモに時刻を記録しておけば、両者の連繋をとることができます。

1. メモをとる
2. 写真を撮る
3. 時刻をメモに記録する
4. 1〜3 を繰り返す

デジタルカメラで撮影した時刻は画像データに保存されるので、メモに記録した時刻と照らし合わせ、適切なところに写真を貼ればよいわけです。

図 1 は、このような方法で作成した講演メモです。講演を聴きながら写真を撮ったりメモをとったりし、並行してメモに時刻を記録しておきます。そして、あとでその時刻にもとづいて写真を貼り込んだものです。この一連の作業が自動的にできれば、さらに便利でしょう。

インストールメモの作成

ソフトウェアを開発したりインストールしたりするときには、試行錯誤を重ねなければならないことがよくあります。本来なら、あとで経緯をみなおせるように頻繁にメモをとったり、実行結果を保存しておいたりすべきなのでしょう。しかし、現実には目の前のことに一所懸命で、それどころではないケースがほとんどだと思います。このような場合、実行内容や出力結果を時刻とともにログにとっておき、時刻を記した簡単なメモもあわせてとっておけば、最小限の手間でインストールメモを残すことができるはずです。

それぞれの時刻において、

- 何を考えて実行したのか
- 何を考えてダウンロードしたのか

といった情報が残っていれば、状況を再現したり、あとで参考にすることがはるかに容易になるはずです。

テキストの場合も、いつ書かれたかが分かるようにしておき、その時刻をもとに別のファイルと関連づけられるようにしておけば、仕事がかなり楽になるのではないでしょうか。

普通のファイルシステムでは、変更を加えたファイルを〝保存した時刻〟は記録されますが、個々の変更がいつおこなわれたかを示す更新時刻は記録に残りません。このような部分テキストの更新時刻も分かるように工夫し、あとで必要になる可能性のあるあらゆる情報を時刻情報付きで保存しておけば、さまざまな処理が楽におこなえるようになるでしょう。

実装

上記のようなシステムの実装は、じつはそれほど難しくはありません。実装手法を以下に順に紹介します。

タイムスタンプの生成

まず、あらゆる情報をタイムスタンプ付きで保存するために、図 2 のような **ta** (TimeAccess) コマンドを作ります。ta に標準入力が与えられた場合は、時刻に応じて、

図 2　ta コマンド

```
#!/usr/bin/env ruby

$: << '/home/masui/lib/ruby'

require 'timeaccess'

ta = TimeAccess.new('/home/masui/TA')

if ARGV.length == 0 then
  File.open(ta.newfile,"w"){ |f|
    f.print $stdin.read
  }
else
  path = File.expand_path(ARGV[0])
  ext = ''
  if path =~ /(\.[^.]+)$/ then
    ext = $1.downcase
  end
  File.symlink(path,ta.newfile+ext)
end
```

図 3　cvs annotate の実行例

```
1.1          (masui    14-Jul-02): //
1.14         (masui    10-Jun-03): //   $Date: 2003/06/05 13:20:36 $
1.14         (masui    10-Jun-03): //   $Revision: 1.13 $
1.1          (masui    14-Jul-02): //
1.1          (masui    14-Jul-02):
1.1          (masui    14-Jul-02): #include <PalmOS.h>
1.3          (masui    15-Jul-02): #include "id.h"
1.14         (masui    10-Jun-03): #include "backup.h"
1.3          (masui    15-Jul-02):
1.6          (masui    18-Jul-02): #define BACKUPDIR "/PALM/Backup"
1.1          (masui    14-Jul-02):
1.1          (masui    14-Jul-02): void Backup();
```

/home/masui/TA/20030812123456

のようなファイルを作成して保存します。引数としてファイルが与えられた場合は、シンボリック・リンクを作成します。

```
% myprog | ta  ←標準出力しつつ時刻ファイルに保存
% ta filename  ←シンボリック・リンクを作成
```

行ごとの更新時刻保存

次に、メモなどのテキストファイルについて、行単位でタイムスタンプを保存するシステムを作ります。各行の更新時刻を保存するには、特別なデータベースを用意する方法も考えられますが、一般的に使われているバージョン管理システム CVS (Concurrent Versions System) を利用するほうが手軽です。

CVS には annotate というコマンドがあり、CVS で管理しているテキストファイルの各行が、どのバージョンにおいて誰によって導入されたのかが分かるようになっています。図 3 は、あるプログラムのソースコードに対して cvs annotate を実行した結果です。

通常の利用形態では、1 行編集するたびに cvs checkin を実行することはないので、変更を加えた複数の行に同じバージョン番号が付きます。しかし、なんらかの変更をおこなうたびに checkin を実行すれば、テキストの各行がどのバージョンから現れたのかが分かるため、結果として、その行の更新時刻が記録に残ることになります。

自動チェックイン

しかし、わずかな編集のたびに cvs コマンドを起動するのは面倒です。そこで、既存のツールを使って cvs を自動的に起動させることにしました。

Emacs では、山岡克美氏が作成した auto-save-buffers.el という Emacs Lisp プログラム[1]を使うと、バッファが定期的にファイルとして保存されるようになります。たとえば、以下のように設定しておけば、編集中のバッファが 0.5 秒ごとに保存されます[1]。

```
(require 'auto-save-buffers)
(run-with-idle-timer 0.5 t 'auto-save-buffers)
```

1 http://www.namazu.org/~satoru/auto-save/auto-save-buffers.el

図 4　CVS 機能付き自動保存

```
;;
;; (require 'auto-save-buffers)
;; (run-with-idle-timer 0.5 t 'auto-save-buffers) ; 0.5秒間隔で保存
;;

;; auto-save-buffers で対象とするファイルの正規表現
(defvar auto-save-buffers-regexp ""
  "*Regexp that matches `buffer-file-name' to be auto-saved.")

(defun auto-save-buffers ()
  "Save buffers if `buffer-file-name' matches `auto-save-buffers-regexp'."
  (let ((buffers (buffer-list))
        buffer)
    (save-excursion
      (while buffers
        (set-buffer (car buffers))
        (if (and buffer-file-name
                 (buffer-modified-p)
                 (not buffer-read-only)
                 (string-match auto-save-buffers-regexp buffer-file-name)
                 (file-writable-p buffer-file-name))
            (progn
              (save-buffer)
              (setq vc-keep-workfiles t)
              (vc-checkin buffer-file-name nil "autosave")
              ))
        (setq buffers (cdr buffers))))))

(provide 'auto-save-buffers)
```

図 5　行の更新時刻を取得するプログラム (linetime)

```
class TimeAccess
    def linetime(path,line)
    dir, file = File.split(path)
    Dir.chdir(dir)
    f = IO.popen("cvs annotate #{file} 2> /dev/null")
    version = f.readlines[line.to_i-1].split[0]
    f = IO.popen("cvs log -r#{version} #{file}")
    date = /date:\s+(\S+\s+\S+);/.match(f.readlines.join)[1]
    t = Time.gm(*ParseDate.parsedate(date))
    t.localtime
    t
  end
end
```

Emacs には、CVS や RCS を扱うための Emacs-VC というバージョン管理パッケージがあります。これを利用し、図 4 のように auto-save-buffers.el にほんのすこし修正を加えれば (下線部)、自動的に cvs checkin をおこなわせることができます。

この結果、0.5 秒間操作がないと自動的に cvs checkin が実行されるため、テキストの各行の更新時刻がほぼ完璧に記録できるようになります。

行の更新時刻の取得

ファイル名と行番号が分かっていれば、cvs annotate を実行し、その行がどのバージョンで導入されたかを知ることができます。しかし、これだけでは正確な時刻は分かりません。

図 6　指定時刻に近いファイルを検索するプログラム

```
class TimeAccess
  def initialize(dir)
    @dir = dir
  end

  def find(cur)
    time = {}
    Dir.chdir(@dir)
    Dir.new(@dir).each { |file|
      next unless FileTest.file?(file) || FileTest.symlink?(file)
      if file =~ /^(\d\d\d\d)(\d\d)(\d\d)(\d\d)(\d\d)(\d\d)/ then
        time[file] = Time.local($1.to_i,$2.to_i,$3.to_i,
                                $4.to_i,$5.to_i,$6.to_i)
      else
        time[file] = File.stat(file).mtime
      end
    }
    res = nil
    time.keys.sort { |a,b|
      time[a] <=> time[b]
    }.each { |id|
      break if time[id] > cur
      res = id
    }
    res
  end
  ……
```

そこで、バージョン番号をもとに cvs log コマンドなどを使ってチェックインした時刻を調べ、その行の更新時刻を取得するようにします（図 5）。

時刻からデータへのアクセス

時刻が指定されたとき、それにもっとも近いファイルは図 6 のようなプログラムで簡単に検索できます。

これらを利用して、時刻から情報を検索するためのプログラム **tafind** を作成します（図 7）。

tafind は、たとえば次のように実行します。

```
% tafind
/home/masui/TA/20030812123456
% tafind 2003/8/18 16:23:45
/home/masui/TA/20030818145203
%
```

linetime（図 5）で取得した時刻を引数に指定して tafind コマンドを起動することにより、それに近い時刻をもつファイルが取得できます。たとえば、あるメモを書いた時刻にもっとも近いタイムスタンプをもつ写真を捜すといったときに使えるでしょう。

Wiki からの利用

ta コマンドで保存したデータを Wiki から呼び出すようにすると便利です。Wiki クローンの 1 つである Hiki[2]では、各種のプラグインが使えます。テキストにメソッド呼出しを記述しておくと、各種の機能を呼び出すことができます。

たとえば、abc.txt というファイルをアップロードしておき、

```
{{attach_src('abc.txt')}}
```

というテキストを文中に書いておくと、abc.txt の内容が〈pre〉〜〈/pre〉に囲まれて表示されます。

同様に、ta で保存したコマンドを呼び出すようなプラグインを作成すれば、

```
{{ta_src}}
```

のような記述により、もっとも近い時刻に保存されたデータを呼び出すことができるようになります。

たとえば、

2 http://www.namaraii.com/hiki/

図 7　tafind

```
#!/usr/bin/env ruby
#
# 指定したファイル(file)のline行目の
# 作成時刻または指定した時刻の直前に
# 作成されたファイルの名前を返す
#
# % tafind file line
# % tafind 2003/8/25 10:23:45
# % tafind
#

$: << '/home/masui/lib/ruby'

require 'timeaccess'
require 'parsedate'

ta = TimeAccess.new('/home/masui/TA')

t = Time.now

if ARGV.length > 1 &&
    File.exist?(ARGV[0]) &&
    ARGV[1] =~ /^\d+$/ then
  t = ta.linetime(*ARGV)
else
  a = ParseDate.parsedate(ARGV.join(' '))
  if a[0] then
    t = Time.local(*a)
  end
end

res = ta.find_path(t)

puts res if res
```

```
% ls | ta
```

を実行したあとで、

```
{{ta_src}}
```

という文字列を含む Hiki ページを作成して書き込むと(図 8)、コマンドの実行結果に展開されて表示されます(図 9)。

システム使用例

これらのシステムと Web を使い、〝夏休みの自由研究〟のようなページを作る例を示します。

たとえば、

- 潮目

図 8　Hiki に書き込むデータ

図 9　実行結果が展開されて表示される

図 10　メモ用のテキストファイル

```
潮目について

はえなわ漁法

イワシの漁獲高推移
```

図 11　各行に関連づけられたファイルを付加したもの

```
潮目について
/home/masui/TA/20030812223956

/home/masui/TA/20030812223956
はえなわ漁法
/home/masui/TA/20030812224044

/home/masui/TA/20030812224044
イワシの漁獲高推移
/home/masui/TA/20030812224140

/home/masui/TA/20030812224140
```

- はえなわ漁法
- イワシの漁獲高推移

に関するページを作ることにしましょう。

まず〝潮目〟について Google で検索すると、福島の潮目に関する解説 (PDF 文書)[3]がみつかりました。これをダウンロードししたあと、ta コマンドで時刻を付けて保存し、メモに「潮目について」と書き入れます。

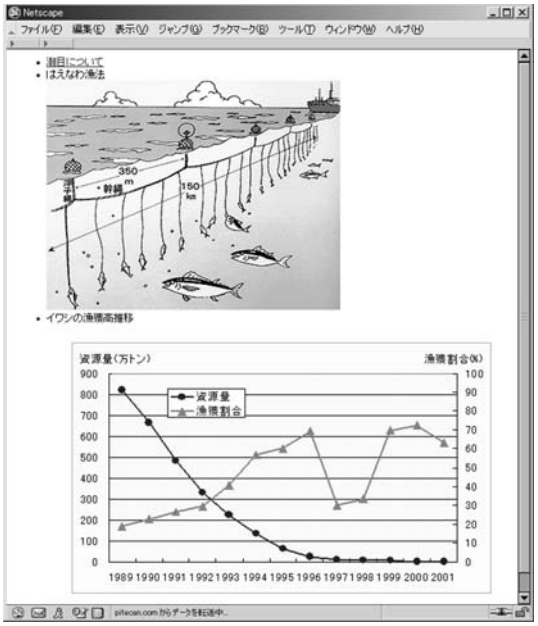
次に〝はえなわ漁法〟について検索すると、今度は那智勝浦のページの解説[4]がみつかったので、この画像をダウ

3 http://www.marine.fks.ed.jp/info_shisetushokai/gareriakagaku.pdf

図 12　HTML に整形

```
<ul>
<li> <a href="http://pitecan.com/TA/20030812223956">
潮目について</a>
<li> はえなわ漁法<br>
<img src="http://pitecan.com/TA/20030812224044">
<li> イワシの漁獲高推移<br>
<img src="http://pitecan.com/TA/20030812224140">
</ul>
```

図 13　得られた Web ページ



ンロードして、同様に「はえなわ漁法」というテキストを書き加えます。

この結果、図 10 のようななんの変哲もないテキストファイルができます。

ところが、前述の cvs を使う方法によって、各行の更新時刻を調べることができるので、それぞれの行に対して tafind を適用すると、図 11 のテキストが得られます。

■　原文：find_path

これを HTML として整形すると、図 12 のような HTML ファイルが得られます。PDF ファイルや画像ファイルの名前をまったく気にせずにデータが活用できることが分かるのではないでしょうか。

4 http://www.town.nachikatsuura.wakayama.jp/mikaku/haenawa.html

おわりに

今回のシステムでは、テキストの修正時刻を記録するのに CVS を利用していますが、この方法だと CVS ファイルが巨大になってしまいますし、実行速度も十分とはいえません。その意味では、得られるメリットとのバランスがとれているとはいいがたいので、実用化するためには実装手法を変えたほうがよさそうです。

これまで、ファイルや情報管理において、時刻情報はあまり重視されていなかったように思えます。「超」整理法や Lifestreams が一時話題になったのも、おそらく時刻情報の活用に注目したという点が斬新だったからでしょう。紙のファイルとは異なり、計算機上では時刻情報をもっと活用できるのではないでしょうか。時刻情報や位置情報など、計算機で簡単に得られるコンテクスト情報をさらに活用する方法を考えていきたいと思います。

（ますい・としゆき　産業技術総合研究所）

[参考文献]

[1] 高林 哲「横着プログラミング (12) とりあげたツールのその後」、UNIX MAGAZINE、2003 年 2 月号