実用的
Ruby
スクリプティング

〈入門から次の段階に進むための スクリプトの書き方講座〉

広瀬雄二◉著

本書籍は、著者の広瀬雄二氏のご好意により下記の条件の基に自由にご利用できます。　2020年5月20日　株式会社カットシステム

広瀬雄二 著『実用的 Ruby スクリプティング』
原作者のクレジット（氏名、作品タイトル）および出版社名を表示し、かつ非営利目的であることを条件に、改変したり再配布したりすることができます。

クリエイティブ・コモンズ
表示 - 非営利

営利目的のホームページでの再配布、公開はご遠慮ください。

本書で取り上げられているシステム名／製品名は、一般に開発各社の登録商標／商品名です。本書では、™ および ® マークは明記していません。本書に掲載されている団体／商品に対して、その商標権を侵害する意図は一切ありません。本書で紹介している URL や各サイトの内容は変更される場合があります。

本書の読み進め方

対象読者

本書はスクリプト言語でのプログラミングに関して、文法的な学習が完了し実用的なプログラムを作る段階に移行したいと考えている読者を主な対象としている。実用的なプログラムといってもパッケージソフトウェアのような大規模なものではなく、「山椒は小粒でひりりと辛い」が当てはまるような、小さいけれども十分に効果を発揮するものの作成を目指すところを想定していて、「スクリプティング」にはそのあたりのニュアンスを込めている。ただし、タイトルにあるようにRubyを使用言語として扱ってはいるが、Rubyそのものの使用技術を高めたり、Rubyの特性を活かしたソフトウェア設計手法を学ぶことなどは主眼としていない。本書は、習う対象であったプログラミングを、実用的なものを作る手段へと転換する時期に必要な知識に触れることを主眼としている。

プログラミング学習の初期の目標はその言語の文法的概念を理解することであるため、学習すべき内容はその言語の中で閉じている。しかし、文法をしっかり覚えても実際に実用的なものが作れるレベルに進むのは意外に難しい。普段使っているソフトウェアのようなものを作れるようになりたいと思うことは、目標設定としては分かりやすいが距離が遠すぎて次にやるべきことが見えてこない。実際にそこに到達するには、データの扱い方、他のプログラムとの連係方法などプログラミング言語の文法体系から離れたところにある知識を吸収していかねばならない。結局のところ、実用的なプログラムが作れるようになるには、プログラムが動作するシステムの作法を覚えていくのがどんな言語を使う場合でも重要である。

本書では、以上のようなことの習得を主眼とするため、Rubyならではの効率的な書き方はあえて避けて記述した部分がある。ただし、他の近代的なスクリプト言語で共通に利用できる便利な概念は、Rubyの場合と断った上で紹介した。もし読者が、プログラミングの学習中で将来この知識を活かそうと思っていたとしても、将来の活躍の場でRubyが使われる確率が100%とは言えない。それは他の言語でも同じことである。重要なのは、その言語とシステムとの関りを意識した無駄のない方法を併せて覚えていくことである。

もう一つ、「実用的なプログラム」を作るために不可欠なこととして、「自作プログラムだけ

にこだわらないこと」が重要である。Unix 系のシステムでは、一つ一つのプログラムの機能を小さくする一方、他のプログラムとのデータの授受のための機能を洗練することで、利用者が複数のプログラムを組み合わせて複雑な処理の流れを自由に構築することが容易になっている。このような構造のプログラムを「フィルタ」という。自分で作るプログラムでも、こうした柔軟な設計の参考とするため、他の代表的なフィルタの利用法も簡単に紹介する。

スクリプティングと KISS の法則

スクリプティングの特徴とは何だろうか。プログラミング言語を用いて何らかのプログラムを作ること全体を「プログラミング」というならば、スクリプティングはプログラミングの中でシンプルかつ簡便さに重きを置くものであると本書は捉える。

一日後に使えることを意識する

「高機能で拡張性も高い」ことがよいプログラムの評価基準であることは確かだが、スクリプティングは犬小屋や整理棚の工作に相通ずる。今後十年に渡って拡張できる崇高な設計にこだわるあまり構想に数週間も掛けているようでは目の前の需要が翌月まで満たせない。まず、必要最低限の機能を最小限のプログラムで作ることを目指す。すぐに楽しめる・すぐに役立つことを重視する。

十年後にも使えることを意識する

上の記述と矛盾するようだが、日常の需要を絶妙に満たしたスクリプトはいつまでも役に立つ。ところがスクリプトの書き方を誤るとスクリプトを動作させる環境の変化によってある日突然動かなくなる。これはシンプルな書き方とシンプルなツールの利用を意識することでぐっと寿命を延ばすことができる。十年前から使い続けられているものは今後十年程度は残り続けるとほぼ考えてよい。

これからプログラミング経験を積んでいく段階では実感し得ないことかもしれないが、自分で作ったプログラムは数日で使命を終えるものから十年以上活躍するもの、自分だけで使うものから多くの人に使ってもらえるものまでさまざまである。最初に作成したものをどんどん成長させていくところにも絶大な楽しみの一つだが、そのときに順調に改良できるかどうかは元のプログラムの見通しのよさにかかっている。

「KISS の法則」という言葉がある。"Keep it simple, stupid!" の略であるこの言葉は、複雑化したがる設計者への戒めであり、実際に作ったものを運用するとき、あるいは修繕や改良など手を入れるとき、複雑なものは手に負えず、結局はシンプルな設計が重要であることを説いたものである。つねに KISS の法則を忘れず、シンプルで見通しのよいものを心掛けたい。

学習の進め方

本書は、一つの題材を解決するために必要な周辺知識をひとまとめにした章割としている。第 1 講には Ruby の習熟前にパラパラと見ると役立つ備忘録的なまとめを載せておいたが、それらの概念的な理解をしていなければ役に立たない。もし、第 1 講を見て内容にピンと来ない場合は Ruby の初学者向けの書籍や Web 資料を併読する必要がある。

プログラミングの学習は外国語の日常会話学習と似ている。文法を覚えるだけでは不十分で、目の前の事象をどれだけ表現できるかは実際に文を組み立てる経験をどれだけ積んだかによって決まる。プログラミング能力向上の早道は「習うより慣れろ」で、自分の体験に優る先生はない。

想定するプログラミング環境

本書では Ruby のバージョン 1.9 系から 2.1 系を前提としているが、1.8 系でも動くように例示プログラムには最大限配慮した。また、各プログラムリストの先頭には

```
#!/usr/local/bin/ruby
```

と記したが、これは読者の環境での Ruby インタプリタのパスに書き換えてほしい。

また、全体として以下のようなソフトウェアの利用を想定している。

種別	本書利用ソフトウェア	前提機能
ウィンドウシステム	X Window System	Tcl/Tk ライブラリが導入されているもの
テキストエディタ	GNU Emacs	保存文字コードとして euc-jp と utf-8 の切り替え可能なもの
仮想端末（ターミナルソフト）	KTerm	表示文字コードとして euc-jp と utf-8 の切り替え可能なもの
シェル	zsh	sh 系のもの（ksh、bash、ash 等）

上記のもの以外でも、表中に示した前提機能を持つものであれば、別のソフトウェアを使用しても構わない。適宜読み換えてほしい。

また、本書では実用スクリプト作成に欠かせない有用なフィルタコマンドとして以下のものの利用を前提としている。

ソフトウェア	説明
NKF（2.0 以上）	ネットワーク用漢字コード変換フィルタ
jless	日本語パッチの当たった less（テキストファイル表示）
grep（egrep）	パターンマッチ行検索
sed	ストリームエディタ
awk	パターン検索・処理言語
sort	テキストファイルの行の並べ替え
wc	行数・単語数・文字数の数え上げ
uniq	重複行の削除と重複数数え上げ
tr	文字種変換
head	ファイルの先頭表示
tail	ファイルの末尾表示

Unix 系の OS や、Mac OS X であればこれらのコマンドは最初から揃っている。ただし NKF に関しては標準添付でない可能性があるため、利用するシステムに相応しい方法で追加インストールする必要があるかもしれない。jless は less の名前のままインストールされている場合もある。日本語表示のできる lv コマンドでも代替できる。

本書で用いる記法

本書では、書体やレイアウトによって以下のような意味付けをしている。

■ コンピュータへの入力

`"gets"` のようなタイプライタ書体で書いた英字は、Ruby プログラムとして意味を持つ単語や値、またはそのようなコマンド名であることを意味する。その他コンピュータに対する入力全般を示す。

■ 省略可能部分

文法的な説明の部分で、

```
if a1 [then]
```

のように [] で括った部分はそれが省略可能であることを示す。上記の例では then という単語をそこに書いても書かなくてもよいことを意味する。

■ 変動値

*文字列*あるいは、*String* のように書いた語は、その場に実際に「文字列」や「String」と書くのではなく、その語で代表される色々な値でそこが置き換えられることを意味する。たとえば「*文字列* .chomp」と書いてある部分は、「*文字列*」の部分が、"foo" や " あいうえお " あるいは、その他の任意の文字列に相当することを意味する。

■ キー入力

本文中に現れる **Return** という部分は、タイプすべきキー操作であることを意味する。Ctrl キーなどの組み合わせについては、以下のように表記する。

C-*x*	Ctrl キーを押しながら *x* を押す。
M-*x*	Meta キーを押しながら *x* を押すか、ESC キーを押して手を離してから *x* を押す。
Return	Return キー（キーボードによっては Enter キー）を押す。
Tab	Tab キーを押す。

■ 実行例と入力部分

段落中に現れる

```
% ./hello.rb
Hello, world!
```

は、対話的な画面での入出力結果を意味する。たとえばコマンド入力行で実行したりエディタ

に入力する例に相当する。例のうち太字で示す部分は人間が入力した部分を意味する。出力結果で特に注目すべき部分を太字で示す場合もある。

■ リスト

段落中に現れる

```
print "ほげほげ"
```

のような部分は、Ruby プログラムのリスト（またはその一部）であることを意味する。

■ 値

リスト中に

```
x
 => [3, 2, 1]
x.reverse
 => [1, 2, 3]
```

のように => 記号に続けて示した部分は直前の変数や式が持っている「値」を意味する。上の例の場合は x という変数の持つ値が [3, 2, 1] であり、x.reverse という文自体が [1, 2, 3] という値を持つ、ということを意味する。それらの値が画面に出たりするわけではない。

また、本文中に現れるバックスラッシュ記号「\」は、日本語キーボードで円マーク「¥」と刻印されているキーを押すと出てくる記号である。コンピュータにとってはどちらの記号も同じだが、プログラミングで使う意味としてはバックスラッシュが適切なのでそちらで表すことにする。

目　次

本書の読み進め方 iii

第 1 講　Ruby 文法のおさらい……1

1.1　プログラム全体 2
1.2　変数 2
1.3　制御構造 3
1.3.1　if －分岐 3
1.3.2　while －条件が成り立つ間の繰り返し 3
1.3.3　case －選択 3
1.3.4　break 4
1.3.5　next 4
1.3.6　redo 4
1.4　比較演算子等 4
1.5　算術演算子 6
1.6　メソッド定義 6
1.7　繰り返し 7
1.7.1　times －回数指定の繰り返し 7
1.7.2　upto と downto －数えながらの繰り返し 8
1.7.3　step －数えながらの繰り返し 8
1.7.4　for －配列要素すべてに対する繰り返し 8
1.7.5　each －配列要素すべてに対する繰り返し 9
1.8　配列 9
1.9　ハッシュ 11
1.10　文字列 12
1.11　ファイル操作 13
1.11.1　データの入力 13
1.11.2　データの出力 14
1.11.3　open メソッドのモード 15
1.12　文字コード指定 15
1.12.1　プログラムファイルの文字コード 15

1.12.2　データファイルの文字コード　16
1.13　正規表現　17
1.13.1　基本表記　17
1.13.2　グルーピングと後方参照　19
1.13.3　最長マッチと最短マッチ　20
1.13.4　正規表現オプション　20
1.13.5　正規表現の文字コード　20
1.14　マニュアルの読み方　21
1.14.1　Ruby オンラインマニュアル　21
1.14.2　メソッド解説の記法　22

第 2 講　入出力処理の基本……25

2.1　入出力処理　26
2.1.1　コマンドライン引数　26
2.1.2　標準入出力　27
2.1.3　標準エラー出力　28
2.2　パターンマッチ処理　29
2.2.1　split による処理　30
2.2.2　正規表現を用いた処理　33
2.2.3　整形出力　35
2.3　簡易データベース処理　38
2.3.1　dbm　38
2.3.2　PStore　43
2.3.3　YAML　47
2.4　排他処理　49
2.4.1　排他処理の必要性　49
2.4.2　Ruby におけるファイルロック　50
2.4.3　排他処理を行なうときの注意　53
2.4.4　共有ロック　54
2.4.5　デッドロック　56
練習問題　58

第 3 講　非対話的処理……59

3.1　メッセージ解析　60
3.1.1　mbox 形式ファイルの解析　60

3.1.2 ヘッダ情報の取り込み 61

3.2 プログラムによるメイル受信 62

3.2.1 電子メイル配送のしくみ 62
3.2.2 dot-qmail のプログラム配送 63
3.2.3 メイル送信コマンド 65
3.2.4 単純返信プログラム 68
3.2.5 メイル自動応答プログラム作成時の注意 69

練習問題 71

第 4 講 プロセスとスレッド……73

4.1 プロセス 74

4.1.1 プロセスの属性 74
4.1.2 fork と exec 75
4.1.3 シグナル 78
4.1.4 シグナル捕捉 79
4.1.5 シグナル捕捉の用例 80

4.2 スレッド 82

4.2.1 Thread の基本的使い方 83
4.2.2 Mutex による競合回避 85
4.2.3 スレッド同士の制御 86
4.2.4 プロセスとスレッドの組み合わせ 89

4.3 外部プログラムとの連係 91

4.3.1 同期呼び出し 91
4.3.2 子プロセスからの出力文字列の受信 92
4.3.3 子プロセスとの双方向通信 93
4.3.4 標準エラー出力を含めたやりとり 97
4.3.5 子プロセス制御プログラム 103

練習問題 107

第 5 講 Ruby らしい記法……109

5.1 プログラミングを手軽にする記法 110

5.1.1 #{ 式 } 110
5.1.2 ` コマンド ` 110
5.1.3 ヒアドキュメント 111
5.1.4 % 記法 112

5.2 Enumerable、Array、Hash......113
5.2.1 collect......114
5.2.2 find......114
5.2.3 select......114
5.2.4 delete......115
5.2.5 delete_if、reject......115
5.2.6 grep......115
5.2.7 sort_by......116
練習問題......117

第 6 講 例外を意識した処理……119

6.1 ファイルとディレクトリ......120
6.1.1 ファイルのクラスメソッド......120
6.1.2 ファイル入出力時の特殊なメソッド......128
6.1.3 ディレクトリ......131
6.1.4 ファイルの消えるタイミング......134
6.2 ファイルテスト演算子......136
6.2.1 test の用法......137
6.2.2 test の利用例......138
6.2.3 すべてのファイルに対しての処理......140
6.3 例外処理......141
6.3.1 例外発生......142
6.3.2 begin 〜 rescue 〜 end......142
6.4 一時ファイルの利用......145
6.4.1 一時ファイルとセキュリティ......146
6.4.2 一時ファイル......147
6.4.3 一時ディレクトリ......147
練習問題......151

第 7 講 周辺ツールの利用……153

7.1 フィルタコマンド......154
7.1.1 nkf －漢字コードの変換......154
7.1.2 egrep －検索パターンにマッチする行を抽出......155
7.1.3 wc －行数・単語数・文字数の表示......156
7.1.4 sed －ストリームエディタ......157

7.1.5 awk －パターンマッチと処理を行なう専用言語......158
7.1.6 sort －入力レコードのソート......160
7.1.7 uniq －重複行の削除と重複数数え上げ......162
7.1.8 tr －文字種変換......163
7.1.9 head －先頭表示......163
7.1.10 tail －末尾表示......164

7.2 複雑な処理を行なう例 164

練習問題 168

■第 8 講 シェルの活用……171

8.1 シェル変数 173

8.1.1 定義と参照......173
8.1.2 条件付き展開......174
8.1.3 特殊な展開......174

8.2 環境変数 175

8.2.1 シェル変数と環境変数......175

8.3 コマンド置換 177

8.4 算術展開 178

8.5 ブレース展開 179

8.6 制御構造 180

8.7 シェル関数 182

8.8 グロッビング 183

8.9 入出力 184

練習問題 187

■第 9 講 自由度の高い CUI……189

9.1 curses ライブラリ 190

9.2 curses を利用してできること 190

9.3 curses の導入 191

9.4 画面制御 192

9.5 即時キー入力 193

9.6 制限時間付きキー入力 194

9.7　文字属性変更 195
9.8　サブウィンドウ 197
9.9　キーパッド 202
9.10　その他必要なメソッド 205
9.11　サンプルプログラム 208
練習問題 211

第10講　GUIプログラミングの基礎……215

10.1　Ruby/TkによるGUIプログラミング 216
10.2　Ruby/Tk 216
10.3　Ruby/Tkの初歩 217
10.4　イベント処理 219
10.4.1　イベントパターンとシーケンス 220
10.4.2　イベントハンドラ 223
10.4.3　command 225
10.4.4　イベントハンドラへの情報 226
10.4.5　即時キー入力処理 227
10.4.6　仮想イベント 227
10.5　ジオメトリマネージャ 228
10.5.1　pack 228
10.5.2　grid 233
10.5.3　place 237
10.6　フレームウィジェット 238
10.7　画像 241
10.7.1　画像のラベルへの貼り付け 241
10.7.2　tkimg拡張ライブラリ 241
10.7.3　画像の手配方法 243
10.8　フォント 246
10.9　代表的なウィジェット 248
10.9.1　ラベル 248
10.9.2　メッセージ 249
10.9.3　ボタン 250
10.9.4　チェックボタン 251
10.9.5　ラジオボタン 252
10.9.6　文字列入力 253

10.9.7　テキストとスクロールバー 255
10.9.8　リストボックス 259
10.9.9　スケール 262
10.9.10　スピンボックス 264
10.9.11　メニュー 265
10.9.12　ダイアログ 269
10.9.13　Canvas 271
10.9.14　Tkc ウィジェット 274
10.9.15　TkcTag 281
10.9.16　バウンディングボックス 281
10.9.17　Canvas 内ウィジェットの検索 281
10.9.18　Canvas ウィジェット使用例 284

10.10 リンク集 288

練習問題 289

練習問題解答 291

あとがき 313

索 引 314

第1講

Ruby 文法のおさらい

この講では、すでに Ruby での簡単なプログラムを作った経験があるという前提で、手短に見直して把握できる程度の説明をまとめておく。

1.1 プログラム全体

ソースプログラムをファイルに記述する上での特徴をまとめる。

- 1 行 1 文で書く。
- セミコロンで区切ると 1 行に複数の文を書ける。
- ブロックの開始と終了は、「do」と「end」、あるいは、「{」と「}」のいずれかで括る。

1.2 変数

- 小文字または _ で書き始める。
- 値には数値、文字列、配列、ハッシュ、など何でも入れられる。
- メソッドの中で定義した変数はそのメソッドの中でだけ有効。
- $ で始まる変数はグローバル変数。
- @ で始まる変数はインスタンス変数で、定義したクラス内全域で有効。
- 大文字で始まるものは定数で、一度代入したら再代入できない。

1.3 制御構造

Ruby の主要な制御構造を示す。

1.3.1　if ー分岐

```
if 条件1 then
  節1
elsif 条件2 then
  節2
  :
  :
else
  節else
end
```

条件 $_1$ が成り立つときは*節* $_1$ を、*条件* $_2$ が成り立つときは*節* $_2$ を、…、いずれでもないときは*節* $_{else}$ を、評価する。

1.3.2　while ー条件が成り立つ間の繰り返し

```
while 条件
  節
end
```

*条件*が真である間*節*を繰り返す。

1.3.3　case ー選択

```
case 値
when 値1[,値1b, ...]
  節1
when 値2
```

```
  節2
else
  節else
end
```

case 直後に書いた*値*を後続する when の後の値と比較し、一致したらそのすぐ後の節を評価する。どの when の値にも一致しなかったときは else 後の節に進む。

1.3.4 break

while などの構文を抜けて次に進む。

1.3.5 next

while などの繰り返し構文の次の回に進む。条件の評価からやりなおす。

1.3.6 redo

while などの繰り返し構文の節先頭に進む。条件の評価はしない。

1.4 比較演算子等

制御構造の条件の部分に利用する比較のための演算子のうち本書で必要なものを示す。

表1.1●比較演算子

演算子	意味
==	左辺と右辺が等しいか
<	左辺が右辺より小さいか

演算子	意味
`<=`	左辺が右辺以下か
`>`	左辺が右辺より大きいか
`>=`	左辺が右辺以上か
`&&`	「かつ」
`||`	「または」
`!`	否定
`not`	否定

条件が成り立つ場合は、「真」を表す **true** が、成り立たない場合は「偽」を表す **false** を返す。なお、Ruby では空を意味する値に **nil** があり、これも「偽」の役割を持つ。したがって、nil でも false でもない値が「真」の役割をする。

`&&`、`||` と同じ意味で、演算優先順位の低い `and` と `or` がある。これはある条件の成否によって処理を行なうかどうかを決める短縮表記として有用である。

条件 `and` *文*　　条件が成り立つときに後続する文を評価する。
条件 `or` *文*　　条件が成り立たないときに後続する文を評価する。

また、条件演算子 ? も文の短縮化に有用である。次の表現全体として値を返す。

条件 ? *式*$_1$: *式*$_2$　　*条件*が成り立つときには*式*$_1$の値を，そうでないときには*式*$_2$の値を返す。

1
2
3
4
5
6
7
8
9
10

1.5 算術演算子

数値の演算は他の言語と共通するものが多い。ただし、C 由来のインクリメント／デクリメント演算子の ++ と -- はない。

表1.2●算術演算子

演算子	意味
+	加算
*	乗算
-	減算
/	除算
%	剰余
**	べき乗
=	通常代入。例：x=5 で x に 5 が代入される
+=	加算代入。例：x+=5 で x が 5 増える
*=	乗算代入。例：x*=5 で x が 5 倍になる
-=	減算代入。例：x-=5 で x が 5 減る
/=	除算代入。例：x/=5 で x が 1/5 になる
%=	剰余代入。例：x%=5 は x=x%5 と同じ
**=	べき乗代入。例：x**=5 は x=x**5 と同じ

1.6 メソッド定義

メソッド定義は以下の形式で行なう。

```
def メソッド(引数リスト)
  定義本体
end
```

1
2
3
4
5
6
7
8
9
10

たとえば、2 つの引数を取るメソッド foo を定義するには次のようにする。

```
def foo(x, y)
end
```

ここで、x と y を**仮引数**と呼び、メソッドを呼ぶときに渡された 2 つの値がその順番で代入される。定義した foo メソッドは次のようにして呼びだす。

```
foo(a, b)
```

なお、メソッドの定義で次のように省略時の値を指定することもできる。

```
def bar(x, y="foo")
end
```

このように定義したメソッド bar は、呼び出し時に第 2 引数を省略することができる。その場合、y の値は "foo" となる。

1.7 繰り返し

繰り返し処理には、数を基準とするものと配列の各要素に対するものの 2 種類がある。

1.7.1　times ―回数指定の繰り返し

```
N.times do
  繰り返し本体
end
```

*繰り返し本体*を *N* 回繰り返す。

1.7.2 upto と downto ー数えながらの繰り返し

```
M.upto(N) do |変数|
  繰り返し本体
end
```

整数 *M* から *N* まで（*M*<*N*）を代入しながら*繰り返し本体*を繰り返す。

```
1.upto(10) do |x|
  printf("%d, ", x)
end
```

は 1 ～ 10 までの整数をすべてカンマ区切りで出力する。

downto は、upto の逆で大きい数から小さい数に向かって繰り返しを行なう。

1.7.3 step ー数えながらの繰り返し

```
B.step(G, S) do |変数|
  繰り返し本体
end
```

初期値 *B* から始めて、*S* ずつ数を増やしながら *G* まで変化する数を*変数*に代入しながら繰り返しを行なう。以下に例を示す。

```
s = 0
1.step(99, 2) do |n|
  s += n
end
printf("1～99の合計は %d です。\n", s)
```

1.7.4 for ー配列要素すべてに対する繰り返し

```
for 変数 in 配列 do
  繰り返し本体
end
```

*配列*の要素を1つずつ取り出し順次*変数*に代入して*繰り返し本体*を繰り返す。

1.7.5 each ―配列要素すべてに対する繰り返し

```
配列.each do |変数|
  繰り返し本体
end
```

for と同じ働きを持つ。

1.8 配列

値の1次元的な集合を表すのが配列である。Rubyでは大括弧内にカンマ区切りで表記する。先頭要素は添字0でアクセスする。Rubyでは配列の要素にどんな種類の値が入っても構わない。

```
[1, nil, "foo", [2, true], 3.14]
```

という配列も可能で、これは第0要素から順に整数の1、nil、文字列の"foo"、配列の[2, true]、浮動小数点数の3.14が格納されている。

配列に備わっている代表的なメソッドには以下のものがある。

表1.3●配列に備わっている代表的なメソッド

メソッド	説明
length	要素数を返す
sort	並べ替え
reverse	逆順化
uniq	重複要素の削除
delete(*val*)	指定要素の削除
shift	先頭要素を取り出して削除

メソッド	説明
unshift(*val*)	要素を配列の先頭に追加
<< *val*	要素を配列の末尾に追加
index(*val*)	指定要素の位置を返す

ブロックを伴うメソッドで、要素を1つずつ変数に代入し、ブロックの評価を繰り返して加工したり集めたりすることができる。

表1.4●ブロックを伴うメソッド

メソッド	説明
each {│*変数*│ *ブロック* }	要素を1つずつ取り出し*変数*に順次代入して*ブロック*を繰り返す
collect {│*変数*│ *ブロック* }	要素を1つずつ取り出し*変数*に順次代入して*ブロック*を評価した値を配列で返す
select {│*変数*│ *ブロック* }	要素を1つずつ取り出し*変数*に順次代入して*ブロック*を評価して真になったときの要素を集めた配列を返す
reject {│*変数*│ *ブロック* }	要素を1つずつ取り出し*変数*に順次代入して*ブロック*を評価して真になったときの要素**以外**を集めた配列を返す
reject! {│*変数*│ *ブロック* }	reject と同様だが元の配列自身を直接操作して該当要素を取り除く

配列要素を一つずつ取り出して繰り返す処理は以下のように each や for を利用する。

```
# eachを使う場合
array.each do |val|
  # valに要素が順に入る
end

# forを使う場合
for val in array do
  # valに要素が順に入る
end
```

1.9 ハッシュ

配列の添字を任意の値にできるものが**ハッシュ**である。ハッシュ値のリテラル表記はブレース（中括弧、{}）を用いる。

```
v = {key1 => value1,
     key2 => value2,
     key3 => value3, ...}
```

これは以下のように代入するのと同じである。

```
v = Hash.new
v[key1] = value1
v[key2] = value2
v[key3] = value3
   :
```

ハッシュの各要素にアクセスするときの記法は配列と同じくブラケット（大括弧、[]）を用いる。

```
x = v[key]
```

ハッシュに備わっている代表的なメソッドには以下のものがある。

表1.5●ハッシュに備わっている代表的なメソッド

メソッド	説明
`length`	キーと値のペアの個数を返す
`keys`	キーのみからなる配列を返す
`values`	値のみからなる配列を返す
`has_key?(`*key*`)`	*key* と一致するキーが存在するか（true か false が返る）
`delete(`*key*`)`	*key* と一致するキーと対応する値を削除する

配列のメソッドでブロックを伴って繰り返すものは、ブロック変数を 2 つにするとハッシュ

で用いることができる。1つ目の変数にはキー、2つ目の変数にはそれに対応する値が同時に代入されブロックが繰り返される。

ハッシュのキーと値を一組ずつ取り出して繰り返す処理は次のように書く。

```
# eachを使う場合
hash.each do |key, val|
  # keyにキー、valに値が順に入る
end

# forを使う場合
for key, val in hash do
  # keyにキー、valに値が順に入る
end
```

ハッシュのキーに存在しないものを指定したときは通常 `nil` を返す。

```
hash["detarame"]             # 存在しないキーだとする
=> nil
```

存在しないキーに対する値は `default=` メソッドで変更できる。

```
hash.default = "ありまへん"
hash["detarame"]
=> ありまへん
```

1.10 文字列

文字列リテラルはダブルクォート、またはシングルクォートで括って表す。

```
# いずれも同じ foo という文字列
"foo"
'foo'
```

ダブルクォートの中では #{ } による値の展開が行なわれる。

```
x = 5
"xの値は#{x}です"
=> xの値は5です。
'xの値は#{x}です'
=> xの値は#{x}です。
```

文字列に対する操作の代表的なメソッドを示す。[ ] 内の添字は 0 から始まる。添字に負の整数を与えると末尾から数える。

表1.6●文字列の操作に対する代表的なメソッド

メソッド	説明
str.length	文字列の長さを返す
str_1+str_2	文字列同士の結合
*str***N*	文字列の *N* 回繰り返し
str[*N*]	文字列の *N* 文字目の取り出し（*str*[0] が先頭）
str[*B*,*L*]	*B* 文字目から *L* 文字分の部分文字列を返す
str[*B*..*E*]	*B* 文字目から *E* 文字目までの部分文字列を返す

1
2
3
4
5
6
7
8
9
10

1.11 ファイル操作

1.11.1　データの入力

Ruby プログラムにデータを与えるには、次の３つの方法がある。

1. gets でコマンドライン引数のファイルを読ませる。

```
% ./program.rb datafile.txt
```

2. gets で標準入力を読ませる。

```
% ./program.rb                      (その1)
データ1
データ2
   :
C-d
% ./program.rb < datafile.txt       (その2)
% cat datafile.txt | ./program.rb   (その3)
```

3. open で特定のテキストファイルを読ませる。

1. と 2. は gets のみで次のような構成で作成する。

```
while line=gets
  ～処理～
end
```

3. は open メソッドを用いて次のような構成で作成する。

```
open(file, "r") do |handle|
  while line=handle.gets
    ～処理～
  end
end
```

1.11.2 データの出力

データをファイルに書き出すには open メソッドのモード指定に "w" を指定する。

```
open(outfile, "w") do |handle|
  handle.print(....)
end
```

次のプログラムは、output.txt ファイルに 1 から 10 までの数字を書き出す。

```
open("output.txt", "w") do |w|
  1.upto(10) do |x|
    w.printf("%d\n", x)
  end
end
```


1.11.3　open メソッドのモード

open メソッドで指定できるモードを以下に示す。このモードは C に近いプログラミング言語ではほぼ共通である。

表1.7●openメソッドのモード

モード	説明
"r"	読み込み用で開く。
"r+"	読み書き両用で開く。読み込みはファイル先頭から行なわれる。
"w"	書き込み用で開く。同名ファイルがあったら空にしてから書き込む。
"w+"	読み書き両用で開く。既存のファイルがあったら空にしてから書き込む。
"a"	書き込み用で開く。既存のファイルがあったら末尾に追加して書き込む。
"a+"	読み書き両用で開く。既存のファイルがあったら末尾に追加して書き込む。

モード指定のアルファベットに "b" を追加するとバイナリモードの指定になり、文字列として扱う場合の文字コード変換処理を無効化する。

1.12　文字コード指定

1.12.1　プログラムファイルの文字コード

Ruby1.9 以降では、プログラム中に英字以外の文字を含ませる場合に、その文字コードを明示する必要がある。プログラムファイルの 1 行目か 2 行目に、次に示すいずれかの書式で文字

コードを記述する。

1. `# coding: Encoding`
2. `# -*- coding: Encoding -*-`
3. `# vim:fileencoding=Encoding`

2. と 3. の方法はテキストエディタに保存文字コードを指定する記法で、それぞれ Emacs、vim 用のものである。Ruby はその記法も解釈するので、利用するテキストエディタに即した記法を利用すると便利である。

Encoding には、用いる文字コード体系に対応して次のいずれかの文字コード名を指定する。

表1.8●Encodingに指定する文字コード

文字コード	文字コード名
UTF-8	utf-8
日本語 EUC	euc-jp
シフト JIS	shift_jis

Ruby が認識する文字コード名一覧は、Ruby 1.9 以降では次のようにして得ることができる。

```
% ruby -e 'p Encoding.name_list'
```

2. または 3. の記法を用いる場合は、得られる一覧のうちテキストエディタの認識する文字コード名と共通のものを選ぶ必要がある。

1.12.2　データファイルの文字コード

これも Ruby 1.9 以降で意識すべき点で、プログラムファイルの文字コードと文字列として読み込むデータの文字コードは原則として統一する。データが別の文字コードの場合は、入出力のときに変換した上で処理を進める。

ファイルからのデータ読み込みでファイルの文字コードが決まっている場合は、open のときのモード指定に変換先文字コード指定を付け加えることで自動変換が行なわれる。たとえば、プログラムファイルが utf-8 で、JIS コード（iso-2022-jp）の入力ファイルを読み込み、なんら

1
2
3
4
5
6
7
8
9
10

かの処理をした結果を euc-jp のファイルに書き出す場合は次のようにする。

リスト1.1●文字コードを指定した処理の例

```
#!/usr/local/bin/ruby
# coding: utf-8
open("jis.txt", "rb:iso-2022-jp:utf-8") do |j|  # JISコードはバイナリモードで
  open("euc.out", "w:euc-jp") do |e|
    while line=j.gets
      print line
      e.print line
    end
  end
end
```

open メソッドのモード指定の後ろにコロンで区切って「外部 Encoding」あるいは「外部 Encoding: 内部 Encoding」の指定を追記でき、外部 Encoding を内部 Encoding に自動変換して読み込んだり、あるいはその逆方向に変換してデータファイルに書き込ませることができる。

1.13 正規表現

1.13.1 基本表記

正規表現は / / で括る。

```
if /正規表現/ =~ 文字列 then
  ～文字列がマッチする場合の処理～
end
```

文字列から正規表現を生成するには Regexp.new を利用する。

```
reg = Regexp.new(正規表現)
if reg =~ 文字列 then
  ～文字列が正規表現にマッチする場合の処理～
end
```

正規表現では、特定の文字を探すための特別な働きを持つメタキャラクタを利用できる。主なものを次に示す。

表1.9●正規表現のメタキャラクタ

メタキャラクタ	説明
.（ピリオド）	任意の 1 字
[*文字クラス*]	選択（下記参照）
*	直前の正規表現の 0 回以上の繰り返し
+	直前の正規表現の 1 回以上の繰り返し
?	直前の正規表現の 0 か 1 の繰り返し
{*N*}	直前の正規表現の *N* 回の繰り返し
{*N*,}	直前の正規表現の *N* 回以上の繰り返し
{*M*,*N*}	直前の正規表現の *M*～*N* 回の繰り返し
^	行頭
$	行末
\w	英数字 [0-9A-Za-z_]
\W	非英数字（\w の逆）
\s	空白文字 [\t\n\r\f]
\S	非空白文字（\s の逆）
\d	数字 [0-9]
\D	非数字（\d の逆）
\b	単語境界
\B	非単語境界

文字クラスの利用例をいくつか示す。

```
[abc123]    a、b、c、1、2、3 のどれか 1 字
[a-z]       a ～ z のどれか 1 字
[^a-z]      a ～ z 以外のどれか 1 字
```

[-0-9]　　ハイフンまたは0～9のどれか1字
[\[\]]　　「[」または「]」のどちらか1字

1.13.2　グルーピングと後方参照

正規表現パターンの一部を（　）で括ると、その部分をひとかたまりにできる。

```
/abcd?efg/
/(abcd)?efg/
```

たとえば前者は？が直前のdだけに作用するが、後者はabcdひとまとめに対して作用する。また、括弧でグルーピングした部分にマッチした文字列は、同じ正規表現内では*数字*、正規表現マッチを抜けた直後では$*数字*で参照できる。*数字*には何番目の括弧かを指定する。たとえば次の例を考える。

```
/(['"]?)[a-z]+\1/ =~ "foo"
/(['"]?)[a-z]+\1/ =~ "'foo'"
```

正規表現自体は「シングルクォートまたはダブルクォートが1回、または0回現れた後ろに、小文字英字が1回以上続き、その後ろにグルーピングの1番目が来る」という意味で、1行目で照合している文字列「foo」の場合は、クォートがないので第1グルーピングでは0回マッチ、つまり空文字なので\1が空文字に置き換えられる。2行目で照合している文字列「'foo'」の場合は、シングルクォートが第1グルーピングでマッチするため、\1はシングルクォート（'）に置き換えられる。

正規表現のマッチングから抜けた直後は$1に空文字列、またはシングルクォートが代入されている。その他、正規表現マッチ直後で利用できる以下の変数も有用である。

$&　　照合文字列のうち正規表現にマッチした部分全体。
$`　　照合文字列のうち正規表現にマッチした部分より前の部分。
$'　　照合文字列のうち正規表現にマッチした部分より後ろの部分。

1.13.3　最長マッチと最短マッチ

メタキャラクタの * と + は、マッチするものが最も長くなるようにマッチングを試みる（最長マッチ）。

```
/Th.*s/ =~ "This is a pen."
=> 0
$&
=> "This is"
```

*? と +? は最短マッチを試みる。

```
/Th.*?s/ =~ "This is a pen."
=> 0
$&
=> "This"
```

1.13.4　正規表現オプション

指定したパターンの照合方式を調整するために正規表現オプションを指定できる。正規表現オプションは /…/ の直後に 1 文字、あるいは、Regexp.new() の第 2 引数に定数を与えて指定する。

表1.10●正規表現オプション

リテラル表記	Regexp.new 第 2 引数	意味
/.../i	Regexp::IGNORECASE	英大文字小文字を同一視する。
/.../m	Regexp::MULTILINE	改行文字も . でマッチする。

1.13.5　正規表現の文字コード

正規表現にも文字コードの概念があり、正規表現と照合対象の文字列の文字コードに食い違いは許されない（Ruby 1.9 以降の場合）。プログラムの記述されたファイルの文字コードが正

規表現の文字コードとなり、照合したい文字列の文字コードと異なるとエラーが発生する。そのような場合は、照合文字列を正規表現の文字コードに変換する（1.12.2、16 ページ）。

Ruby マニュアル正規表現[注1] も参照。

1.14 マニュアルの読み方

Ruby の基本的な文法・用法を理解したのちは、

- アルゴリズムの組み立て方に習熟する
- ライブラリの存在と利用方法を覚える

という段階に進む。**ライブラリ**とは、すでに完成しているある程度機能が揃ったメソッド（関数）の集合体のことで、プログラム作成時に利用することができる。Ruby では標準セットの中に多くのライブラリが入っている。プログラムで何の前準備もなく使えるものを組み込みライブラリといい、Hash（ハッシュ）や Array（配列）、String（文字列）もそれらの 1 つである。

1.14.1　Ruby オンラインマニュアル

Ruby リファレンスマニュアル[注2] には、Ruby のすべての仕様が記述されている。プログラミングを行なうときにはつねに開いておくべきページである。

Hash や Array は特に重要なクラスで、付属しているメソッドをしっかり把握して挙動を確かめておきたい。これらは、上記 Web ページの「組み込みライブラリ[注3]」の一覧に含まれている。このマニュアルを見て各種クラスで使えるメソッドなどを理解して使用する方法を覚えていく。

注 1　http://doc.ruby-lang.org/ja/2.1.0/doc/spec=2fregexp.html
注 2　http://doc.ruby-lang.org/ja/2.1.0/doc/index.html
注 3　http://doc.ruby-lang.org/ja/2.1.0/library/_builtin.html

たとえば Array クラス[注4]の説明ページを見よう。クラスの説明ページには

- スーパークラス
- インクルードしているモジュール
- クラスメソッド
- メソッド

の項目がある。

スーパークラスはそのクラスの親となるクラスで、親クラスに備わっている性質はすべて子となるクラスも持つことになる。したがって、そのクラスで使えるメソッドのすべてを知るにはスーパークラスのメソッドも調べる必要がある。

クラスメソッドはインスタンスを作ることなく使えるメソッドである。一方、クラスメソッドでないメソッドは生成したインスタンスからのみ使える。たとえば、Array クラスのクラスメソッド new は、新しい Array オブジェクトを生成するためのもので、これはクラス名直属で使う。

```
x = Array.new(5)
```

この例では新しいオブジェクトが変数 x に代入された。x に代入されている Array クラスの具体的実体のことを**インスタンス**という。

一般メソッドはインスタンスそのものに対して働く。Array クラスの例でいえば、push メソッドはそのインスタンス（Array の場合配列値）に別の要素を追加する。

```
x.push("Hello")
```

1.14.2　メソッド解説の記法

メソッドの見出しは次のようになっている。

```
Array.new([size[, val]])
```

注 4　http://doc.ruby-lang.org/ja/2.1.0/class/Array.html

大括弧［ ］はその中味が省略可能であることを意味する。また、書体の変わっている *size*、*val* は、実際にはさまざまな値が来る変動値であることを意味する。したがって、上記の 1 行は次のいずれにも該当する。

```
Array.new()
Array.new(5)
Array.new(i, 4)
```

第2講

入出力処理の基本

2.1 入出力処理

なんらかの入力に対して加工を施し、結果を出力する。形こそ違えソフトウェアは原則としてそのような動きを持つ。Rubyで作成するコンソールプログラムでの入出力の方法をまとめる。

2.1.1 コマンドライン引数

Ruby プログラム起動時に、そのプログラム名の後ろに指定された文字列は単語ごとに区切られた配列として変数 ARGV に入る。

```
% ./program.rb a "b c" d\ ef ghi
```

としたときの ARGV は次のようになる。

```
ARGV[0] = "a"
ARGV[1] = "b c"
ARGV[2] = "d ef"
ARGV[3] = "ghi"
```

コマンドラインに指定した文字列の単語分解はシェルが規則に基づいて行なっている。

Ruby の場合はコマンドライン引数の個々をファイル名と見なし、そのファイルの中味全体を結合したファイル入力を自動的に利用することができる。これは変数 ARGF[注1] を介して利用する。コマンドライン引数が何もない場合の ARGF は標準入力を指すオブジェクトとなる。

gets メソッドを単体で使用することは ARGF からの読み込みを意味する。つまり、次の２つのコードは同じ処理を行なう。

```
while line=gets
  ...
end
```

注 1　http://doc.ruby-lang.org/ja/2.1.0/class/ARGF.html

```
while line=ARGF.gets
  ...
end
```

2.1.2　標準入出力

Rubyプログラムに引数を何も与えない場合、単体で呼ぶ`gets`メソッドは標準入力（STDIN、Standard Input）から読み込む。また、単体で呼ぶ`puts`、`print`、`printf`メソッドは、標準出力（STDOUT、Standard Output）に書き出す。標準入力と標準出力は、特に指定がなければ端末に結び付けられる。つまり、入力はキーボード、出力は仮想端末の画面となる。

これらは、シェルの持つリダイレクションやパイプの機能で入出力先をファイルや別のプログラムに切り替えることができる。

`>` *file*	標準出力の行き先を*file*に切り替える。
`<` *file*	標準入力の供給元を*file*に切り替える。
$program_1$ ｜ $program_2$	$program_1$の標準出力を$program_2$の標準入力につなぐ。

いくつか例を示す。

```
# program.rbの出力結果を outfile に
% ./program.rb > outfile

# program.rbへの入力を infile から与える
% ./program.rb < infile

# ls -l の出力結果を program.rb に送る
% ls -l | ./program.rb

# infileの内容を program.rb に渡し、その出力を wc -l に渡し、
# その出力を outfile に保存(下記いずれも同じ)
% < infile ./program.rb | wc -l > outfile
% ./program.rb < infile | wc -l > outfile
% ./program.rb < infile | > outfile wc -l
% < infile ./program.rb | > outfile wc -l
```

Ruby プログラムでは、起動時に引数を与えるとそれをファイルと見なし、単体の gets はそこからデータを読む。起動時に引数を与えた場合でも必ず標準入力を読みたい場合は、明示的に標準入力を表す STDIN（あるいはそれが代入された変数 $stdin）から gets する。

```
while line = STDIN.gets
  ～処理～
end
```

2.1.3 標準エラー出力

標準出力は処理した結果を出力するためのものである。処理に失敗したときなどのエラーメッセージは、標準エラー出力に出すべきである。簡単な例を示す。

```
print "整数を2で割った数を求めます。偶数を入れてください: "
n = gets.to_i
if n % 2 == 0 then               # 2で割った余りが0なら(偶数)
  printf("%d\n", n/2)            # 結果のみ出力
else
  printf("%dは偶数ではありません。\n", n) # エラーメッセージ出力
end
```

このプログラムをファイルへの出力リダイレクション付きで起動し、奇数を入力すると、次に示すようにエラーメッセージはファイルに書き込まれて画面には表示されない。

```
% ./div2.rb > outfile
5
% cat outfile
整数を2で割った数を求めます。偶数を入れてください: 5は偶数ではありません。
```

　原則として、プログラムの処理結果は次の処理の入力として利用するかもしれないので、エラーメッセージなどはそれらと分けて標準エラー出力に出力した方がよい。そうすれば、処理結果をリダイレクトしても、利用者向けのメッセージはそれに混ざることなく画面に表示される。

　Ruby で標準エラー出力に出力するには、STDERR（あるいは $stderr）を使う。前述の例

を STDERR を使って書き直したものを次に示す。

```
STDERR.print "整数を2で割った数を求めます。偶数を入れてください: "
n = gets.to_i
if n % 2 == 0 then              # 2で割った余りが0なら(偶数)
  printf("%d\n", n/2)           # 結果のみ出力
else
  STDERR.printf("%dは偶数ではありません。\n", n) # エラーメッセージ出力
end
```

先ほどと同じように実行した例を次に示す。

```
% ./div2.rb > outfile
整数を2で割った数を求めます。偶数を入れてください: 5
5は偶数ではありません。
% cat outfile
%
# 何もない。結果がないことになるのでこれでよい。
```

2.2 パターンマッチ処理

与えられたデータを自動的に処理するプログラムの必要性と有用性は高い。そのことを見越して、自動処理しやすい形式でデータファイルを作成しておくことが重要である。また、Web ページのアクセスログなど、計算機が自動的に生成するログファイルはテキスト形式で書き出されるものが多い。

テキストを処理するプログラムは、その入力パターンに応じて処理を決定する。入力パターンの照合は対象となる書式を元に決める。

- カンマ「,」やタブ文字などの特定文字列による分解（例：CSV ファイル）
  ⇒ split メソッドで分解

● 特定キーワードで分解
⇒　正規表現を用いて解析

2.2.1　split による処理

ある特定の文字でフィールドを区切られている書式（DSV、Delimiter-Separated Value）で、フィールド値に区切り文字が現れないようなものは split で簡単に解析できる。CSV ファイルの処理例を示す。

yamagata.csv[注2] は、文字列もクォートなしで書かれた CSV ファイルで、

第 6 フィールド　　市町村名
第 9 フィールド　　総人口
第 10 フィールド　　男人口
第 11 フィールド　　女人口

となっている。

リスト2.1●yamagata.csv（一部）

```
7,大項目,地域コード,地域識別コード,境域年次(2010),境域年次(2000),,人口　総数,人口　男,人口　女,世帯数
総数,世帯数　一般世帯,世帯数　施設等の世帯
11,,6201,2,2010,2000,山形市,254244,121433,132811,96560,96425,135
12,,6202,2,2010,2000,米沢市,89401,43953,45448,33013,32920,93
13,,6203,2,2010,,鶴岡市,136623,64846,71777,45514,45395,119
20,,6204,2,2010,,酒田市,111151,52610,58541,38955,38860,95
25,,6205,2,2010,2000,新庄市,38850,18432,20418,12980,12958,22
26,,6206,2,2010,2000,寒河江市,42373,20497,21876,12717,12702,15
27,,6207,2,2010,2000,上山市,33836,16036,17800,10751,10709,42
28,,6208,2,2010,2000,村山市,26811,12846,13965,7865,7860,5
29,,6209,2,2010,2000,長井市,29473,14211,15262,9269,9228,41
30,,6210,2,2010,2000,天童市,62214,30148,32066,20404,20338,66
31,,6211,2,2010,2000,東根市,46414,22934,23480,14388,14343,45
```

注 2　e-Stat 政府統計の総合窓口（http://www.e-stat.go.jp/SG1/estat/List.do?bid=000001034995cycode=0）表番 2. 男女別人口及び世帯の種類（2 区分）別世帯数都道府県、市部、郡部、市町村・旧市町村より抜粋

```
32,,6212,2,2010,2000,尾花沢市,18955,9138,9817,5332,5320,12
33,,6213,2,2010,2000,南陽市,33658,16025,17633,10567,10538,29
```

これを split で分解して、市町村と総人口のみの集計出力を得るには以下のようにする。

リスト2.2●split-yg.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-

while line=gets
  data = line.chomp.split(",")
  if /\d+/ =~ data[7]       # 第8フィールドが数字の連続なら
    printf("%s,%s\n", data[6], data[7])
  end
end
```

実行結果を次に示す。

```
% ./split-yg.rb yamagata.csv
山形市,254244
米沢市,89401
鶴岡市,136623
酒田市,111151
新庄市,38850
寒河江市,42373
上山市,33836
村山市,26811
長井市,29473
天童市,62214
東根市,46414
尾花沢市,18955
南陽市,33658
```

このように、CSV やタブ文字区切りなどで明確にフィールド分けされたテキストファイルは split メソッドで必要なフィールドを抽出できる。

ただし、CSVファイルでも文字列をダブルクォートで括ったり、なおかつその内部にさらにカンマやダブルクォートが含まれるようなものは split で単純にフィールド分けできない。たとえば次のようなCSVファイルが該当する。

リスト2.3●quoted.csv

```
番号,氏名,ローマ字,所属
1,公益太郎,"KOEKI, Taro",野球部
2,飯森花子,"IIMORI, Hanako",茶道部
3,三川一二三,"MIKAWA, Hifumi",SKIP
```

クォートを含むCSVを読んで加工し、CSVに書き出す処理は、csvライブラリ[注3]を利用する。これを利用して上記のCSVファイルをフィールドごとに分解してみる。

リスト2.4●csv-split.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'csv'                   # csvライブラリの利用

CSV.foreach("quoted.csv") do |row|
  puts row.join("|")            # 縦棒(|)でフィールドを区切って出力
end
```

フィールド値にカンマが含まれていても、クォートされているため一つのフィールド値として処理されていることが以下の実行例から分かる。

```
% ./csv-split.rb
番号|氏名|ローマ字|所属
1|公益太郎|KOEKI, Taro|野球部
2|飯森花子|IIMORI, Hanako|茶道部
3|三川一二三|MIKAWA, Hifumi|SKIP
```

注3　http://doc.ruby-lang.org/ja/2.1.0/library/csv.html

2.2.2　正規表現を用いた処理

固定区切りでないものは正規表現でパターンマッチを行ない、後方参照を用いてマッチした一部を取り出す。

以下のような電子メイルのファイルから差出人を抽出してみる。

リスト2.5●1217.txt

```
Return-Path: <ta01002@e.koeki-u.ac.jp>
Delivered-To: yuuji@itl.koeki-u.ac.jp
Received: (qmail 12673 invoked by uid 1010); 26 Apr 2014 22:52:17 -0000
Received: (qmail 12659 invoked from network); 26 Apr 2014 22:52:17 -0000
Received: from broy.e.koeki-u.ac.jp (HELO localhost) (172.17.54.112)
  by pan.e.koeki-u.ac.jp with SMTP; 26 Apr 2014 22:52:17 -0000
Received: from broy.e.koeki-u.ac.jp (HELO localhost) (172.17.54.112)
  by pan.e.koeki-u.ac.jp (antibadmail 1.38) with SMTP; Apr 27 07:52:17 JST 2014
Date: Mon, 27 Apr 2014 07:52:10 +0900 (JST)
Message-Id: <20140427.075210.193698131.ta01002@e.koeki-u.ac.jp>
To: yuuji@e.koeki-u.ac.jp
Subject: Nice to meet you
From: HIROSE Yuuji <ta01002@e.koeki-u.ac.jp>
Mime-Version: 1.0
Content-Type: Text/Plain; charset=iso-2022-jp
Content-Transfer-Encoding: 7bit
Status:

はじめまして、こんにちは、
では、さようなら。

おしまい。

--
公益太郎
```

差出人はメイルヘッダの Return-path か、From の値から取得する。電子メイルのヘッダは、1 行目から始まり、空行で終わる。行頭（先頭カラム）から始まり、コロンまでのものがフィールド名、その後ろがフィールド値である。

メイルのヘッダ部分のみを繰り返す処理は次のように書ける。

```
while line=gets
  break if /^$/ =~ line    # 行頭(^)の直後に行末($)で空行を表すパターン
  ～処理～
end
```

この間で、Return-path または From を得る処理は次のようにする。

```
while line=gets
  break if /^$/ =~ line
  if /^(return-path|from): *(.*)/i =~ then
    from = $1
  end
end
```

これを応用して、1 通のメッセージを含むファイルから差出人などの情報をサマリ表示するプログラムを次に示す。

リスト2.6●headervalue.rb

```
#!/usr/local/bin/ruby
# coding: euc-jp
# Parse rfc5322 header and display summary

while line=gets
  break if /^$/ =~ line
  if /^(return-path|from): *(.*)/i =~ line then
    from = $2
  elsif /^subject: (.*)/i =~ line then
    subj = $1
  elsif /^date: (.*)/i =~ line then
    date = $1
  end
end

printf("%s: %sさんからの「%s」というメイルです。\n", date, from, subj)
```

この例では、Return-Path と From に加え、サブジェクトを保持する Subject ヘッダと、日付けを保持する Date ヘッダの値をそれぞれ取得している。

2.2.3 整形出力

プログラムで集計処理した結果を仮想端末画面に出すようなものの場合、出力項目ごとに桁揃えした形式にすると読みやすくなる。たとえば CSV ファイルの出力例として、

```
% ./csv-split.rb
番号|氏名|ローマ字|所属
1|公益太郎|KOEKI, Taro|野球部
2|飯森花子|IIMORI, Hanako|茶道部
```

のようなものがあったが、以下のようにすると見やすくなる。

```
% ./csv-split.rb
番号|氏名            |ローマ字                 |所属
   1|公益太郎        |KOEKI, Taro              |野球部
   2|飯森花子        |IIMORI, Hanako           |茶道部
   3|三川一二三      |MIKAWA, Hifumi           |SKIP
```

このような場合、printf のフォーマット指定で桁幅を指定すればよい。各フィールドの桁幅を以下のように見積もる。

第 1 フィールド　　4 桁右寄せ
第 2 フィールド　　16 桁左寄せ
第 3 フィールド　　25 桁左寄せ
第 4 フィールド　　指定なし（残り桁すべて）

これを表現する printf フォーマット指定は次のようになる。

```
printf("%4s|%-16s|%-25s|%s\n", *row)
```

csv-split.rb（リスト 2.4）の出力部分を上の printf に書き換える。

リスト2.7●csv-printf.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'csv'                    # csvライブラリの利用

CSV.foreach("quoted.csv") do |row|
  printf("%4s|%-16s|%-25s|%s\n", *row)
end
```

ところがこれを実行すると以下のような結果が得られる。

```
番号|氏名                |ローマ字                         |所属
   1|公益太郎            |KOEKI, Taro                   |野球部
   2|飯森花子            |IIMORI, Hanako                |茶道部
   3|三川一二三          |MIKAWA, Hifumi                |SKIP
```

これは、%s では日本語 1 字が 1 とカウントされるのに対し、実際に表示される桁幅が 2 であるため、表示幅計算がずれるためである。

Ruby 1.8 以前で euc-jp や shift_jis コードを用いた場合は、日本語 1 字の幅が 2 と計算されたためずれることなく表示された。Ruby 1.9 以降や utf-8 を利用する場合で、日本語でのきれいな桁揃えを行ないたい場合は、文字列を一度 euc-jp のような 2 バイトコードに変換し、なおかつバイト文字列と Ruby に見なさせてフォーマットを行なわせ、出力のときに元の文字コードに戻すようにするとよい。具体的には以下のプログラムを利用する。

リスト2.8●kprintf.rb

```
class String
  require 'kconv'
  if defined?("".force_encoding)
    def toeucbin()
      self.toeuc.force_encoding("binary")
    end
  else
```

```
    def toeucbin()
      self.toeuc
    end
  end
end

class IO
  def printf(*args)
    out = sprintf(*(args.collect{|x| x.is_a?(String) ? x.toeucbin : x }))
    print out.toutf8
  end
end

class Object
  def printf(*args)
    if args[0].is_a?(String)
      $stdout.printf(*args)
    else
      port = args.shift
      port.printf(*args)
    end
  end
end
```

1
2
3
4
5
6
7
8
9
10

日本語桁揃え処理をしたいプログラムの冒頭で、上記 kprintf.rb を読み込むように変えると printf の桁揃えがうまくいく。

リスト2.9●csv-kprintf.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'csv'                   # csvライブラリの利用
require './kprintf.rb'          # 日本語桁幅揃え(この行を追加)

CSV.foreach("quoted.csv") do |row|
  printf("%4s|%-16s|%-25s|%s\n", *row)
end
```

実行すると以下のように桁が揃う（等幅フォントの場合）。

```
% ./csv-kprintf.rb
番号|氏名          |ローマ字                |所属
   1|公益太郎      |KOEKI, Taro             |野球部
   2|飯森花子      |IIMORI, Hanako          |茶道部
   3|三川一二三    |MIKAWA, Hifumi          |SKIP
```

本講では、日本語を含む整形出力処理が必要なプログラムではこの kprintf.rb を読み込むようにする。

2.3 簡易データベース処理

レコードの集合を永続的に保存させ、いつでも取り出せ、さらに加工ができるようにするシステムをデータベース管理システム（DBMS）という。実用されている DBMS は単純なデータの保存、取り出しだけでなく検索や資源管理などを豊富な機能を含むシステムである。

DBMS の利用は、作成するプログラムで扱えるデータの規模を飛躍的に増大させられるという利点もあるが、そのためには DBMS 上でのデータ構造の構築方法について一定量学習する必要がある。最終的な目標は DBMS 下で管理するデータの利用を置きつつ、ここでは Ruby プログラムから容易に利用できるシンプルなデータベース機構を取り扱う。データの取り出しだけでなく、追加や更新機能を付けたい場合はこれらのものが有用となる。

2.3.1 dbm

Unix システムの多くは dbm という汎用的なデータベースライブラリを備えている。簡易データベースはこれで十分作成できる。Ruby では添付ライブラリの dbm[注4] を介して簡単に利用できる。

注 4　http://doc.ruby-lang.org/ja/2.1.0/library/dbm.html

dbmによるデータベースはHash[注5]と同様にkeyとvalueの対の集合をデータとする。ただし、Hashと違い、keyとvalueがともに文字列（String）でなければならない。

dbmは次の形式で利用し、「*変数*」と「*dbm ファイル*」を結び付ける。

```
DBM.open(dbmファイル) do |変数|
  ...
end
```

この変数はハッシュのように利用でき、do 〜 endブロックを抜けると自動的に変数の値がdbmファイルに保存される。

ここではCSVで書かれたデータを読み込んでdbmに追加する例を示す。CSVのレコードは第1フィールドをキー（重複するものがないと保証できる）として扱うことにする。

リスト2.10●dbm-add.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'dbm'                    # dbmクラスを利用
datafile = "./database"          # データベース名

# CSVデータを読んでハッシュに入れておく
data = Hash.new
while csv=gets
  # 第1フィールドをキー、残りをごっそり文字列として値にする
  if /([^,]*),(.*)/ =~ csv
    data[$1] = $2
  end
end

# ndbm形式のデータベースを開く
DBM.open(datafile) do |x|
  # 開いた時点で既にデータがあれば x に入った状態でスタートする
  for key, val in data           # data の key, val ペアを取り出して繰り返す
    x[key] = val
  end
end
```

注5　http://doc.ruby-lang.org/ja/2.1.0/class/Hash.html

このプログラムを用いて、データ登録をしてみる。以下のような 2 つに分けた CSV データを用意する。

リスト2.11●yama-1.csv

```
山形市,Yamagata-shi,山形市旅篭町
米沢市,Yonezawa-shi,米沢市金池
鶴岡市,Tsuruoka-shi,鶴岡市馬場町
酒田市,Sakata-shi,酒田市本町
新庄市,Shinjo-shi,新庄市沖の町
寒河江市,Sagae-shi,寒河江市中央
```

リスト2.12●yama-2.csv

```
上山市,Kaminoyama-shi,上山市河崎
村山市,Murayama-shi,村山市中央
長井市,Nagai-shi,長井市ままの上
天童市,Tendo-shi,天童市老野森
東根市,Higashine-shi,東根市中央
尾花沢市,Obanazawa-shi,尾花沢市若葉町
南陽市,Nanyo-shi,南陽市三間通
```

```
% ./dbm-add.rb yama-1.csv
% ls -l data*
-rw-r--r--  1 yuuji  wheel  16384 Feb 12 10:45 database.db
% strings data*
(中味を確認)
% ./dbm-add.rb yama-2.csv
% ls -l data*
% strings data*
```

(n)dbm 形式のデータファイルは `makedbm -u` でテキスト部分を抽出できる。

Solaris の場合

```
% makedbm -u database
```

1
2
3
4
5
6
7
8
9
10

NetBSD の場合

```
% makedbm -u database
```

FreeBSD の場合

```
% yp_mkdb -u database
```

Linux では、NIS サーバパッケージをインストールすると makedbm コマンドが使用できるものが多い。もし、makedbm コマンドが見付からない場合は、以下のスクリプトで代用できる。

リスト2.13●dumpdbm.rb

```
#!/usr/local/bin/ruby
require 'dbm'
DBM.open(ARGV[0]){|d| d.each{|i| printf("%s\t%s\n", *i)}}
```

dumpdbm.rb では、データベースファイルの拡張子を省略して起動する。

作成したデータベースを読み込むプログラムも簡単で、DBM.open を使えばよい。

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'dbm'                   # dbmクラスを利用
datafile = "./database"         # データベースファイル名

# ndbm形式のデータベースを開いて順次レコードを出力
DBM.open(datafile, 0666, DBM::READER) do |x|
  # DBM.open(datafile) だけで開いてもよいが、
  # 読み込みだけのときは第3引数を DBM::READER とする
  for key, val in x             # x の key, val ペアを取り出して繰り返す
    printf("%s -> %s\n", key, val)
  end
end
```

データ追加、ダンプ、検索の機能を選べるようにしたプログラム例を示す[注6]。

注6　dbm-ops.rb を実行してみよ。

リスト2.14●dbm-ops.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'dbm'                   # dbmクラスを利用
datafile = "./database"         # データベースファイル名

if ARGV[0] == nil then
  STDERR.puts "#{$0} -a [Data.csv]      Add record"
  STDERR.puts "#{$0} -d                 Dump database"
  STDERR.puts "#{$0} -s                 Search on database"
  exit 1
end

case ARGV.shift
when "-a"                       # データの追加登録
  # CSVデータを読んでハッシュに入れておく
  DBM.open(datafile) do |x|
    # DBM.openしてからデータ入力を行なう処理は好ましくない(後述)
    while csv=gets
      # 第1フィールドをキー、残りをごっそり文字列として値にする
      if /([^,]*),(.*)/ =~ csv
        x[$1] = $2
      end
    end
  end
when "-d"                       # 一覧出力
  DBM.open(datafile, 0666, DBM::READER) do |x|
    for key, val in x
      printf("%s -> %s\n", key, val)
    end
  end
when "-s"                       # 検索
  STDERR.print "検索キー: "
  kwd = STDIN.gets.chomp!       # STDIN無しだと -s というファイルから読む
  reg = Regexp.new(kwd, nil, "n")   # "n" 文字コード変換なしで検索
  DBM.open(datafile, 0666, DBM::READER) do |x|
    require 'kconv'
    for k, v in x
      if reg =~ k then          # マッチしたレコードのみ出力
        printf("%s -> %s\n", k, v)
```

```
      end
    end
  end
end
```

ただし、このプログラムの追加登録部分は頻繁なデータ更新が必要な場合に効率低下を招く潜在的な問題を含んでいる。それについては 2.4 節で説明する。

データのキー削除には delete メソッドを用いる。

```
db.delete(key)
```

とすると該当するキーと値を削除できる。

ユーザ名とパスワードなど、キーと値の対で管理する類のデータは dbm が向いている。dbm は、それがインストールされているシステム上では同一のファイル形式であることが保証される。このため、別の言語やツールとデータを共有する必要がある場合には dbm 形式を用いるのが有利である。ただし、同じ種類の dbm をインストールしている場合でも、計算機の種類が違うとデータに互換性がない場合があることに注意する。

2.3.2 PStore

Ruby 固有の汎用的なデータ永続化クラスが PStore[注7] で、ほぼすべてのオブジェクトがファイルに保存できる。また、dbm と違い一つのデータファイルに複数の Ruby オブジェクトを格納できる。

PStore を使う場合の基本的な流れは以下のとおりである。

```
require "pstore"
db変数 = PStore.new(保存ファイル)
db変数.transaction do
  # db変数を利用した処理
end
```

注 7　http://doc.ruby-lang.org/ja/2.1.0/class/PStore.html

*db 変数*はほぼハッシュと同様に利用でき、添字にキーを指定するとそれに対応する値を取得できる。PStore ではキーのことを「ルート名」と表現し、*db 変数*［*ルート名*］の形で、*ルート名*に対応する任意のオブジェクトにアクセスでき、そのオブジェクトは transaction 終了時にファイルに保存される。したがって、*db 変数*に代入された値はその後の起動でも利用できることになる。

ここでたとえば、ハッシュと配列の 2 つの集合値を保存したい場合を考える。以下の例は、

- ルート名 " 山のデータ " でハッシュ値
- ルート名 " 偶数 " で配列値

をそれぞれ格納し、データファイル ps-data に保存するものである。

リスト2.15●ps-add.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'pstore'
datafile = "ps-data"            # データを保存するファイル名
# 初期値のハッシュをPStoreファイルに保存
# このプログラムを3回実行したのち ps-list.rb を実行する

db = PStore.new(datafile)
db.transaction do
  m = db["山のデータ"] = db.fetch("山のデータ", Hash.new)
  # これでmが保存可能なハッシュになる
  m["富士山"]   = 3776
  m["月山"]     = 1984
  m["鳥海山"]   = 2236

  e = db["偶数"] = db.fetch("偶数", Array.new)
  # これで e が保存可能な配列になる
  e << 2
  e << 4
  e << 6       # 2、4、6を順次配列に追加する
end
```

リスト 2.15 のプログラムを 3 回実行後、リスト 2.16 のプログラムを実行してみる。

リスト2.16●ps-list.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'pstore'
datafile = "ps-data"

# datafileに登録されているものを取り出す。
db = PStore.new(datafile)
db.transaction do
  m = db["山のデータ"] = db.fetch("山のデータ", Hash.new)
  e = db["偶数"] = db.fetch("偶数", Array.new)
  # 既存データがある場合でも同じ代入方法でよい
  puts("山のデータのハッシュ")  # mを順次出力
  for k, v in m
    printf("%s\t-> %4d\n", k, v)
  end
  puts("偶数の配列")            # mを順次出力
  puts e.join(", ")
end
```

```
% ./ps-list.rb
山のデータのハッシュ
月山     -> 1984
富士山   -> 3776
鳥海山   -> 2236
偶数の配列
2, 4, 6, 2, 4, 6, 2, 4, 6
```

このように、前回の値が残ったまま代入されるので、配列には値が積み重なってゆく。

「商品」と「単価」の key、value ペアを持つハッシュを永続的に持ち、追加で増やせるプログラム例を示す。

リスト2.17●pstore-basic.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require './kprintf.rb'
require 'pstore'
datafile = 'pricedb'

def showall(hash)
  for k, v in hash
    printf("%-20s%4d円\n", k, v)
  end
end

db = PStore.new(datafile)
db.transaction do
  price = db["price-list"] = db.fetch("price-list", Hash.new)
  while true
    STDERR.print "商品=単価　の形式で入れてください(C-dで終了): "
    break if (line=gets) == nil
    redo if /([^=]*)=([\d ]*)/ !~ line # マッチしなかったら redo
    price[$1] = $2.to_i
  end
  puts "\n全商品リストです"
  showall(price)
end
```

PStoreはプログラム終了時の変数の値をまるごと持ち越せるため、細かいことを特に意識しなくてもデータを永続させることができる。ただし、現状ではRuby固有のものであるため、Ruby以外の処理系とのデータのやりとりはできない。また、Rubyの内部構造に依存したデータをファイルに書き込むため、データファイルを保存したときのRubyとバージョンの異なるRubyでは読み取れないこともある。必要に応じて、CSV形式など他のテキスト形式のファイルに書き出すプログラムなどを作成しておくことも考慮した方がよい。

なお、PStoreなどの機構を利用して、プログラム動作中の変数の値を論理的な構造を保ったままファイルのようなバイト列に変換することを**シリアライズ**という。

2.3.3　YAML

PStore とほぼ同様の使い勝手・機能で利用でき、保存データファイルの汎用性の高いものがYAML[注8]である。YAML はさまざまな言語で構造化されたデータをシリアライズするために策定されている仕様で、Ruby からも利用できる。ただし、言語の垣根を越えて利用することが主眼であるため、保存できるデータの種別が以下のものに限られている。

マッピング	ハッシュ／辞書（Ruby では Hash に対応）
シーケンス	配列／リスト（Ruby では Array に対応）
スカラ	文字列や数値等

スカラとして保存できるのは、整数、浮動小数点数、文字列、日付、真偽値で、このうち文字列は utf-8 コードのもののみ扱える。

値のファイル保存に YAML を用いるには、yaml/store ライブラリを用いて次のような流れで行なう。

```
require "yaml/store"
db変数 = YAML::Store.new(保存ファイル)
db変数.transaction do
  # db変数を利用した処理
end
```

PStore の使用例と比べて分かるように、変数と保存ファイルを結び付けた後の使用方法は全く同じである。PStore の例で示したリスト 2.15 と同じ機能を YAML を用いて記述すると以下のようになる。

リスト2.18●yaml-add.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'yaml/store'
datafile = "data.yaml"          # データを保存するファイル名
# ハッシュをYAML形式ファイルに保存
```

注 8　http://yaml.org、http://docs.ruby-lang.org/ja/2.1.0/library/yaml.html

```
db = YAML::Store.new(datafile)
db.transaction do
  m = db["山のデータ"] = db.fetch("山のデータ", Hash.new)
  # これでmが保存可能なハッシュになる
  m["富士山"]   = 3776
  m["月山"]     = 1984
  m["鳥海山"]   = 2236

  e = db["偶数"] = db.fetch("偶数", Array.new)
  # これで e が保存可能な配列になる
  e << 2
  e << 4
  e << 6       # 2、4、6を順次配列に追加する
end
```

太字部分が PStore からの変更点で、プログラムの根幹は全く変更なしで動くことが見て取れる。PStore での保存ファイルは人間が直接読み書きできないバイナリ形式だが、YAML 形式の保存ファイルは可読性が高い。上記の yaml-add.rb を動かして生成されたファイルを見てみよう。

```
% ./yaml-add.rb
(ここでYAMLデータファイルを確認)
% cat price.yaml
---
"山のデータ":
  "富士山": 3776
  "月山": 1984
  "鳥海山": 2236
"偶数":
- 2
- 4
- 6
```

YAML 形式では、マッピング（Hash）はキーと値をコロンで区切ったもの、シーケンス（Array）はハイフンで始まる行の並びで表現される。この書式を守ってファイルを直接編集することで、データの追加や削除を手動で容易に行なうことができる。

なお、YAML でデータの並びを行ごとに記述する方式をブロックスタイルという。改行によらず

```
{キー1：値1，キー2：値2}
[要素1，要素2，要素3]
```

のようにそれぞれマッピング、シーケンスを表現する方式をフロースタイルという。

データのファイル保存を YAML で行なうのは効率の面ではあまり有利でないが、保存されたデータを直接確認してプログラム修正の参考としたり、Ruby 以外で作成された他のツールに、構造を保ったままの複雑なデータを渡したりできるなどの利点がある。

2.4 排他処理

データの読み書きを行なうプログラムは、いつどのようなタイミングで起動されても問題が起きないように注意して作成する必要がある。どのような問題の可能性をどのように回避するかについて考える。

2.4.1　排他処理の必要性

複数のプログラムが 1 つのデータファイルにある情報を利用して処理した結果を書き戻すという処理をほぼ同時に行なうと整合性が取れなくなる。たとえば、あるファイルに書かれている数値を 1 だけ増やすプログラムを 2 つ走らせたとする。最初、ファイルの内容が

```
10
```

だったとして、1 増やすプログラムを 2 回動かすと 12 になるはずである。しかし以下のように、「プログラム起動その 1」が発生したわずか 10 ミリ秒後に「プログラム起動その 2」が起きた場合を考える。

経過時間（ミリ秒）	プログラム起動その 1 の流れ↓	プログラム起動その 2 の流れ↓
0	起動開始	
10	データファイルを開く	起動開始
20	10 を読み取る	データファイルを開く
30	変数に 1 を足す	10 を読み取る
40	ファイルに 11 を書き込む	変数に 1 を足す
50	終了	ファイルに 11 を書き込む
60		終了

「プログラム起動その 2」の実行主体がデータを読み取るタイミングでもまだ値は 10 なので、最終的な書き込み結果は 11 になる。

このように、あるファイルの内容を読み取ってなんらかの修正を加えるプログラムは、同じプログラムが同時進行している可能性を考慮して、次に示すような注意を払う必要がある。

- 他のプロセスがそのファイルを処理しているときはそれが終わるまで待つ。
- 自プロセスがファイルを open している時間を極力短くする。

同時に 2 つのプロセスが同じ処理を行なわないようにすることを**排他処理**という。ファイル修正の排他処理には**ファイルロック**が用いられる。

2.4.2 Ruby におけるファイルロック

Ruby では File クラス[注9]の `flock` メソッドでロックを行なう。

前述の例のように、データファイルに記録された数値を 1 増やすプログラムを考えてみる。ここでは DBM 形式データベースに「10」と書いたものを用意し、その後任意の整数を足していくプログラムを考える。

まず DBM 形式データファイルに `"count" => "10"` というハッシュ値を格納する。以下のようなコマンドラインで作成できる。

```
% ruby -rdbm -e 'DBM.open("counter"){|c| c["count"]=10}'
# 以下のように確認
```

注 9　http://doc.ruby-lang.org/ja/2.1.0/class/File.html

```
% makedbm -u counter
count 10
```

この値を増やしていくプログラムとして以下のものを使用する。

リスト2.19●count-plus.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
# ロックなしのまずい例

dbfile = "counter"
require 'dbm'
DBM.open(dbfile) do |c|          # 変数cのハッシュがDBMファイルに直結
  count = c["count"].to_i        # 文字列なので整数に変換
  STDERR.printf("現在の値は%dです。いくつ足しますか: ", count)
  count += STDIN.gets.to_i       # 読み込んだ数を加算
  c["count"] = count             # データを更新して終了
end
```

1つだけ起動して動きを確認する。ここでは10を足してみる。

```
% ./count-plus.rb
現在の値は10です。
いくつ足しますか: 10
% makedbm -u counter
count 20
```

続いて、ファイルロックがないために起きる不整合の例を示す。端末ウィンドウを2枚開き、2つのコマンドラインで同時に起動してみる[注10]。

［端末その1］

```
% ./count-plus.rb
現在の値は20です。
いくつ足しますか: 1
```

［端末その2］

```
% ./count-plus.rb
現在の値は20です。
いくつ足しますか: 2
```

注10 dbmライブラリの実装によっては、2つめ以降のプログラムが異常終了させられる。

両プログラムを起動してから、各ウィンドウで整数を入力する。すると、あとで終わらせた方の加算値になっていることが分かる（次の結果は端末２への値の入力をあとでやった場合)。

```
% makedbm -u counter
count 22
```

ファイルロック処理を追加してみる。ロックの必要な処理の前後に、ロック設定処理とロック解除処理をそれぞれ入れる。流れとしては次のようになる。

```
lockfile="ロックファイルの名前"
open(lockfile, "w") do |lf|
  lf.flock(File::LOCK_EX)     # ロック設定
  DBM.open(datafile) do |x|
      :
  end
  lf.flock(File::LOCK_UN)     # ロック解除
end
```

このような修正を行なった count-flock.rb を示す。

リスト2.20●count-flock.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
# ロック処理を追加 (ただしロック時間が無駄に長いのでのちに改善)

dbfile = "counter"
lockfile = dbfile + ".lck"      # ロックに使うダミーファイル
require 'dbm'
open(lockfile, "w") do |lf|
  lf.flock(File::LOCK_EX)       # ロック設定
  STDERR.puts "#{dbfile} に対する作業開始"
  DBM.open(dbfile) do |c|       # 変数cのハッシュがDBMファイルに直結
    count = c["count"].to_i
    STDERR.printf("現在の値は%dです。\nいくつ足しますか: ", count)
    count += STDIN.gets.to_i    # 読み込んだ数を加算
    c["count"] = count          # データを更新して終了
  end
  STDERR.puts "作業終了"
```

```
  lf.flock(File::LOCK_UN)
end
```

このプログラムを2つの端末で同時に起動する。

［端末その1］
先に起動して、入力待ちになったら端末その2でプログラムを起動し、その2の方で先に値を入れてから、その1に戻って値を入れる。

```
% ./count-flock.rb
現在の値は22です。
いくつ足しますか:
```

［端末その2］
その1での起動直後にこちらも起動し、入力ガイドが出る前に値を入力する。

```
% ./count-flock.rb
3
```

［端末その1］
その2で値入力後、その1でも値を入れると処理が完了する。

```
いくつ足しますか: 5
作業終了
```

［端末その2］
その1での更新処理が完了すると同時にこちらも処理が完了する。

```
counter に対する作業開始
現在の値は27です。
いくつ足しますか: 作業終了
```

このプログラムでは flock を行なうファイルとしてダミーファイルを open したが、これは排他処理を行なう必要のあるファイル形式が dbm だからで、普通に open して修正を行なうファイルの排他処理を行なうなら、そのファイルを open したオブジェクトで flock すればよい。

PStore と YAML は transaction ブロックそのものが排他処理を行なうため、flock などの明示的操作はしなくても構わない。

2.4.3　排他処理を行なうときの注意

先述のとおり flock 処理は極力短くする。さもなくばロック解除待ちのプロセスが溢れかえる。`count-flock.rb`（リスト2.20）は、加算すべき値の入力処理もロック内で行なってい

るので、入力をすぐに行なわないとずっとロックがかかったままとなる。データベース更新処理のみをロックするように書き換える。データ更新部分だけを抜き出したプログラム count-flock2.rb を示す。

リスト2.21●count-flock2.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
# ロック処理を追加 (ロック時間短縮版)

dbfile = "counter"
lockfile = dbfile + ".lck"      # ロックに使うダミーファイル
require 'dbm'

STDERR.print "いくつ足しますか: "
x = STDIN.gets.to_i             # 加算値を先に読み込んでおく
open(lockfile, "w") do |lf|
  lf.flock(File::LOCK_EX)       # ロック設定
  STDERR.puts "#{dbfile} に対する作業開始"
  DBM.open(dbfile) do |c|       # 変数cのハッシュがDBMファイルに直結
    count = c["count"].to_i
    STDERR.printf("現在の値は%dです。", count)
    c["count"] = count+x        # データを更新して終了
  end
  STDERR.puts "作業終了"
  lf.flock(File::LOCK_UN)
end
```

2.4.4 共有ロック

データの更新を伴う処理は次の流れで行なう。

1. 元のデータの読み込み
2. データの修正
3. データベースへの書き込み処理

　このとき、必要な修正が元データの値に影響される場合は、上記の一連の処理全体をロックし、修正手続中のデータベースが他のプロセスからアクセスできないようにする必要がある。
　一方、データベースを読み込むだけの処理もあるとする。この場合、

- 読み込み処理の途中で他のプロセスに修正処理をされると困る（データの不整合が起きる可能性がある）。
- 読み込み処理の途中で他のプロセスが読み込み処理をするのは問題ない。

という関係となる。このため、修正処理と、読み取りのみの処理が混在する場合、読み取りのみの処理を**共有ロック**として処理する。
　共有ロックはFile::LOCK_SHを指定する。
　まとめると以下のようになる。

- 排他ロック（File::LOCK_EXで指定）
  処理対象に他のロックが全く掛かっていない場合のみロックする。
- 共有ロック（File::LOCK_SHで指定）
  処理対象に他の排他ロックが掛かっていない場合のみロックする。既存の他のロックが共有ロックのみならロックする。

　同じDBMデータファイルに対して共有ロックをかけ、読み取り処理のみを行なうプログラムcount-shlock.rbを示す。

リスト2.22●count-shlock.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
# 共有ロック処理（共有ロックのプログラムだけなら同時に何個でも起動できる）

dbfile = "counter"
lockfile = dbfile + ".lck"       # ロックに使うダミーファイル
require 'dbm'
open(lockfile, "w") do |lf|
  lf.flock(File::LOCK_SH)        # 共有ロック設定
  STDERR.puts "#{dbfile} に対する作業開始"
```

```
  DBM.open(dbfile) do |c|
    count = c["count"].to_i
    STDERR.printf("現在の値は%dです。\n", count)
  end
  sleep 10              # 同時起動の実験をするため敢えて時間をかける
  STDERR.puts "作業終了"
  lf.flock(File::LOCK_UN)
end
```

2つの端末を開き、以下の組み合わせで起動して動きを確かめよ。

パターン	端末1での操作	端末2での操作
1	(1)count-flock.rbを起動 (4) 値を入力	 (2)count-flock.rbを起動 (3) 値を入力
2	(1)count-flock.rbを起動 (3) 値を入力	 (2)count-shlock.rbを起動
3	(1)count-shlock.rbを起動	 (2)count-flock2.rbを起動 (3) 値を入力
4	(1)count-shlock.rbを起動	 (2)count-shlock.rbを起動

（端末1で`count-shlock.rb`を起動した場合（パターン3、4）は、10秒以内に端末2での操作を完了させる。難しい場合は`count-shlock.rb`のsleep値を延長すればよい。）

2.4.5　デッドロック

複数のプロセスが互いに別プロセスのロック解除を待ち続けて永遠に解除が起きない状態を**デッドロック**という。

たとえば、2つのファイルAとBがあり、それぞれに数値が書かれていたとする。このとき

- プロセス1→「Aファイルの値+1」をBに書き込む

- プロセス 2 →「B ファイルの値 +1」を A に書き込む

という 2 つのプロセスがほぼ同時に走ったとする。プロセス 1 は「A をロックしてから B をロック」し。プロセス 2 は「B をロックしてから A をロック」しようとしたとする。もし、「プロセス 1 が A をだけをロック、プロセス 2 が B だけをロック」した状態になったら、この状態は永遠に解除されない。

この例の場合、デッドロックを回避するには複数の資源をロックする順位を決めておく。A と B であれば、A → B という順番でのみロックするように設計する。

大規模なソフトウェアではロックすべき資源は多数あるので、デッドロックを完ぺきに回避するためには入念な設計が必要である。全体的に可能性を極力下げるようにするための方策として以下のものが挙げられる。

- 複数資源を同時にロックしない
- 同時ロックが必要なら順番を決める
- ロックしている時間が最短になるよう工夫する

これらに注意するのと同時に、デッドロック回避が本質的に難しいことを考慮して、

- ロックしなくていいアルゴリズムに変える
  - レコードごとにファイルを分けたりファイルの存在そのものをレコードとして利用する
  - 同時アクセスが起きたらエラーにしてよい処理はエラーにする
- デッドロックが起きた場合のエラー処理を作り込んでおく
  - 時間を置いてリトライ
  - タイムアウトを決めて異常停止として処理を打ち切る

などを作り込んでおくとよい。

練習問題

2.1 コマンドラインに与えた単語（群）をファイル名として、そのファイル（群すべて）の中味を出力するプログラム cat.rb を作成せよ。

2.2 起動するたびに日付と文章の入力をそれぞれ求め、PStore データベース（diary.pstore）に追加格納していくプログラム diary1.rb を作成せよ。完成物の実行例は以下のようになる。

```
% ./diary1.rb
日記の日付を入力してください(YYYY-MM-DD): 2014-04-17
文章を入力してください(終了は行頭で C-d):
今日は初ツーリング!
でも意外に寒くて十六羅漢でギブアップ、そしてあったかとびうおラーメン!
C-d
% ls -l diary.pstore
-rw-r--r--  1 yuuji  wheel  161 Apr 17 10:41 diary.pstore
```

2.3 前問の日記プログラムで生成した diary.pstore に格納された日記文章を一覧表示するプログラム diary2.rb を作成せよ。完成物の実行例は以下のとおり。

```
% ./diary2.rb
【2014-04-17】
今日は初ツーリング!
でも意外に寒くて十六羅漢でギブアップ、そしてあったかとびうおラーメン!
  :
  :
(以下登録した分だけ続く)
```

2.4 前問で作成した diary.pstore ファイルを、YAML 形式に変換するプログラム ps2yaml.rb を作成せよ。

第3講

非対話的処理

3.1 メッセージ解析

これまで作成したプログラムはコマンドラインで手で起動していた。そのような対話的利用のプログラムだけでなく、ネットワーク先からのアクセスによる起動や、人手によらない自動起動が前提となるプログラムの構成を考えてみる。最も身近なものとして送信された電子メイルをきっかけに起動するプログラムを考えてみよう。

電子的に取り扱う色々なファイルのうち、ネットワークごしにやりとりするもののほとんどは一定の書式を持つテキストファイルである。電子メイルもその一つで人間の読める単純な形式で送受信が行なわれている。この形式を解析し、その情報にふさわしい挙動をするようなプログラムを作ってみよう。

3.1.1 mbox 形式ファイルの解析

電子メイルのファイルフォーマットは RFC5322[注1] によって規定されている。これが格納されたファイルを通称 mbox 形式といい、おおまかなルールは以下のようになっている。

- 1 行目から空行までがヘッダ（header）、そのあとに本文（body）が来る。
- ヘッダはフィールド名とそれに対応する値がコロン区切りで並ぶ。
- ヘッダは 1 行 1 レコードだが、行頭が空白で始まるときは前の行のフィールド値の続きと見なす。

簡単な例を示す。

```
From: KOEKI Taro <taro-k@e.koeki-u.ac.jp>
Subject: This is
 an example
To: IIMORI Hanako <hanako@f.koeki-u.ac.jp>
Date: Sat May  9 11:43:26 JST 2014

こんにちは...(以下本文続く)
```

注 1　RFC5322 Internet Message Format - https://www.ietf.org/rfc/rfc5322.txt、日本語訳は http://srgia.com/docs/rfc5322j.html 等参照。

3.1.2　ヘッダ情報の取り込み

mbox 形式のファイルを読み取り、ヘッダ部分の値を取り出すプログラムを考える。各フィールド名（上の例の場合 From、Subject、To、Date）とその値をハッシュに格納する。ヘッダ解析のループは以下の方針で作成する。

- 空行ならヘッダの終端なのでヘッダ解析終了。
- 行頭が空白なら前の行のフィールド値に現在の行を追加する。
- （空白でなければ）フィールド名と値に分解しハッシュに代入する。
- 以上を繰り返す。

これを Ruby で記述した一例を示す。

```
header = Hash.new
field=""
while line = gets
  break if /^$/ =~ line                # 行末ならヘッダ解析終了
  if /^(\s)+(.*)/ =~ line then         # 行頭空白なら
    header[field] += $1+"\n"+$2        # 前回のフィールド値に追加
  elsif /^([^:]+):\s*(.*)/ =~ line     # フィールド定義のパターン
    header[field=$1] = $2              # 初期値として代入
  end
end
```

なお、このプログラムは同じフィールド名が複数回現れる場合の考慮はしていない。

1
2
3
4
5
6
7
8
9
10

3.2 プログラムによるメイル受信

たとえば、特定の宛先にメイルを送ると商品の資料が返送されるサービスがある。これはもちろん人間が送り返しているのではなく、プログラムによって自動的に返送処理がされている。どこかから発信された電子メイルが届くとき、そのメッセージがファイルとして保存されるだけではなく、指定したプログラムを起動し、メッセージの内容を標準入力として読み取らせることもできる。この仕組みを利用し、メイル配送時に自動的に起動され、受信メッセージを読み取りつつ動くプログラムを作成してみよう。

3.2.1 電子メイル配送のしくみ

電子メイルは、送信者がクライアントソフトウェアを用いてメッセージを送ると、送信者側のメイルサーバを経て受信者側のメイルサーバに届く（下図参照）。

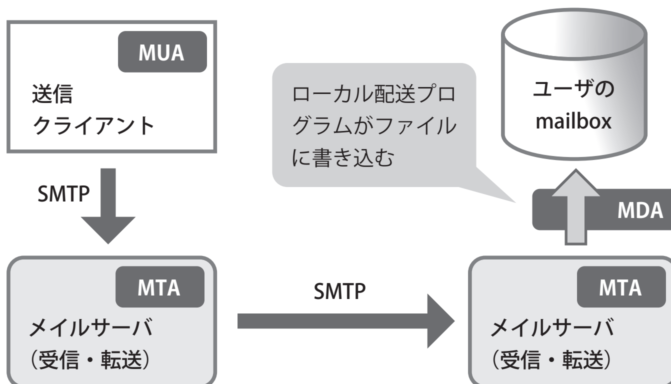


図3.1●電子メール配送のしくみ

いくつかのメイルサーバを経由する場合もあるが、最終的には受信者のメイルボックスのあるメイルサーバで、ファイル単位の書き込み処理がされる。

ここでは、MDA（Mail Delivery Agent）が、送られたメッセージをファイルに書き込むときに自動処理を行なわせる方法を考える。本講ではメイル配送下でプログラムを動かすときの制

1
2
3
4
5
6
7
8
9
10

御の自由度の最も高いqmailをローカル配送プログラムに利用する例を示す。qmailでは拡張アドレス機能によって個人ユーザのメイルアドレスを何個でも作れ、なおかつ配送時に起動するプログラムに対して十分な情報を環境変数に設定する機能があるためスクリプト作成が非常に効率的に行なえる。

なお、近年PC-Unix環境でデフォルトのMTAとして搭載されることの多いPostfixが稼働しているメイルサーバでも、qmailとほぼ同様の拡張アドレス機構と環境変数設定を可能にするdot-qmail deliver program[注2]を使うことで以下の説明がそのまま適用できる。個人権限で導入できる単一のシェルスクリプトだが、qmailと同様の拡張ルールが使えるようになるため、メイルを自在に操るプログラムを作成する際には導入を検討する価値がある。

以後の説明は、qmailを前提とした記述となっているが、Postfixの配送に上記スクリプトを組み合わたものでも適用できる。

3.2.2　dot-qmailのプログラム配送

一般的なメイルサーバプログラムでは、ユーザ宛てメッセージの処理を「ファイルへの保存」、「他のアドレスへの転送」、「コマンド起動」のいずれかから選べる。qmailの場合、一般ユーザ宛メッセージの最終配送先をどうするかは~*user*/`.qmail`ファイルの内容で決定する。このファイルはdot-qmail(5)ファイル形式で記述する。1行1エントリで以下のいずれかを何個でも書ける。

表3.1●dot-qmail(5)ファイル形式

行頭文字	種類	例
#	コメント	# メモ
|	プログラム	|./program arg
. または / （スラッシュで終わらない）	mbox形式のファイル	./mbox
. または / （スラッシュで終わる）	maildir形式のディレクトリ	./maildir/
&	転送メイルアドレス	&taro@example.ac.jp

また、dot-qmailファイルとして~*user*/`.qmail-`*ext*があると、*user*-*ext*という宛先の配送先として利用する。これを**拡張アドレス**といい、*ext*の部分を拡張子という。また、*ext*を

注2　http://www.gentei.org/~yuuji/software/dotqmail/

"default" にした ~*user*/.qmail-default というファイルを作ると、対応する dot-qmail ファイルがない場合のデフォルトの宛先となる。つまり、*user*-abc、*user*-xyz、*user*-hoge、*user*-foo、…など任意の拡張子を持つローカルな宛先に対して、対応する dot-qmail ファイルがない場合はすべて ~*user*/.qmail-default で定義される宛先に配送される。

dot-qmail ファイル中の「|」で始まる行は、すぐ後ろに書いたプログラムをその記述のとおりに起動する。このとき以下の環境変数が自動的に設定された状態でプログラムが呼ばれる。

表3.2●プログラムが呼ばれるときに設定される環境変数

環境変数	値の意味
SENDER	エンベロープ sender
RECIPIENT	受信者アドレス
USER	受信者のユーザ名
HOME	受信者のホームディレクトリ
HOST	受信アドレスのドメイン部
LOCAL	受信アドレスのローカル部
EXT	拡張子部分（ユーザ名以降最初のハイフン以降）
EXT2	$EXT の 1 個目のハイフン以降
EXT3	$EXT の 2 個目のハイフン以降
EXT4	$EXT の 3 個目のハイフン以降
DEFAULT	default でマッチした文字列

たとえば、以下のような送信が行なわれた場合を考えよう。

差出人のアドレス　hanako@example.com
宛先のアドレス　taro-foo@example.co.jp

受取側の taro-foo@example.co.jp のうち「-foo」は拡張子で、受信者となるユーザは taro である。この場合の配送先は ~*taro*/.qmail-foo ファイルで決定され、そこに次のように書いてあったとする。

```
| program args...
```

以上の条件で *program* が起動するときの環境変数は、次のように設定される。

表3.3●前記の例で設定される環境変数

環境変数	値
SENDER	hanako@example.com
RECIPIENT	taro-foo@example.co.jp
USER	taro
HOST	example.co.jp
LOCAL	taro-foo
EXT	foo

注意すべきはRECIPIENT変数の値で、これは実際に配送される宛先のアドレスになるのであって、メッセージに書かれているTo: アドレスの値になるとは限らない。たとえば、

```
To: hogehoge@example.co.jp
Cc: taro-foo@example.co.jp
```

のように出されたメッセージでも、taro-fooに届くときはRECIPIENT=taro-foo@example.co.jpになる。

上記の環境変数をうまく利用すればメイル処理プログラムが効率的に作れる。簡単な例として、送信者に「ありがとう」とだけ返すプログラムを作る。

メイルアドレスは*user*-39で作る。この配送先は~/.qmail-39ファイルに書き込んで作成する。

```
% echo '| ./pf3/thankyou.rb' > ~/.qmail-39
```

ホームディレクトリから見て./pf3/thankyou.rbの名前で送信者（$SENDER）にメイルを送るRubyプログラムを作る。その前にメイルを送信するコマンドについて調べる。

3.2.3　メイル送信コマンド

メイルを送信するための原始的なプログラム**sendmail**を利用すると、自動的なメイル送信が容易に行なえる[注3]。

注3　qmailやPostfixにも同名の互換コマンドが用意されている

```
% sendmail -f from@add.ress to@reci.pie.nt...
```

これで標準入力を読んだ結果を送信する。ヘッダと本文を正しく入れて送信してみる。

```
% sendmail -f 自分のアドレス 自分のアドレス
To: 自分のアドレス
From: 自分のアドレス
Subject: test テスト

Hello!
はろー!
C-d
```

実際に自分のアドレスに届いたメイルを確認してみよう。使用するソフトウェアにもよるが、規格どおりに作ってある正しいソフトウェアでは以下のように化けた内容になるはずである。

```
Subject: test 繝??せ繝

Hello!
縺??繧阪!
```

メイルのヘッダや本文に日本語を入れたい場合は、どの文字コードを使うかを宣言する宣言文を入れた上で、その文字コードに正しく変換したものを送出しなければならない。ヘッダのフィールド値に日本語を入れたい場合はMIMEエンコードする必要がある。また、本文に入れる場合はヘッダ内に利用する文字コードの形式を入れる。本文でJISコード（ISO-2022-JP）を用いる場合は、以下のような記述をヘッダに追加する。

```
MIME-Version: 1.0
Content-type: text/plain; charset=iso-2022-jp
```

文字コードはnkfライブラリ[注4]で変換できる。元の文字列をヘッダの値向けにMIMEエンコードするには

注4　http://doc.ruby-lang.org/ja/2.1.0/class/NKF.html

```
NKF.nkf('-jM', String)
```

とし、本文向けに JIS コードに変換するには

```
NKF.nkf('-j', String)
```

とする。日本語 Subject 付の日本語メッセージを送る簡単なプログラムは以下のようになる。

リスト3.1●sendjpex.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-

require 'nkf'
subj = '日本語サブジェクト'
body = 'こんにちは、さようなら。'

if ARGV[0] == nil then
  STDERR.puts "送り先アドレスを指定してください。"
  STDERR.puts " 例: #{$0} toaddress@example.jp"
  exit 1
end

command = sprintf("| sendmail %s", ARGV[0])
header = sprintf("To: %s
Subject: %s
Content-type: text/plain; charset=iso-2022-jp
Mime-Version: 1.0\n\n", ARGV[0], NKF.nkf('-jM', subj))
open(command, "w") do |mail|
  mail.print header
  mail.print NKF.nkf('-j', body)
end
```

以下のように起動して日本語込みのメッセージが届くことを確認しよう。

```
% ./sendjpex.rb 送信先@アドレス
```

3.2.4 単純返信プログラム

元の課題に戻ろう。~/.qmail-39 に指定したプログラムで、「送信者」に「ありがとう」とだけ返すプログラムを作成してみる。「送信者」つまり送り返すべき宛先はスクリプト起動時の環境変数 SENDER から取得し、そこに送信するのが簡単である。

プログラムは以下の流れで作成する。

- メイルが届くと ~/.qmail-39 に書かれたとおりにプログラムが自動的に起動される。このとき標準入力を読むと送信メッセージの内容を 1 行目から順に得ることができる。
- プログラム起動のきっかけとなったメイルの「送信者」のアドレスが環境変数 SENDER に入っているので、これを自動返信メイルの宛先とする。
- プログラム起動のきっかけとなった受信アドレスが環境変数 RECIPIENT に入っているので、これをスクリプトから送るメイルの送信者アドレスとする。
- Subject は MIME エンコードし、本文は JIS コード変換してから送る。

環境変数は、Ruby の変数 ENV にハッシュとして入っている。たとえば、ENV["FOO"] は、環境変数 FOO が定義されている場合はその値が、定義されていない場合は nil が返る。

リスト3.2●thankyou.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'nkf'

sender = ENV['SENDER']          # 環境変数SENDERの値の取得
rcpt   = ENV['RECIPIENT']       # 環境変数RECIPIENTの値の取得

if sender == nil || rcpt == nil then
  STDERR.puts "$SENDER and $RECIPIENT not set. exit."
  exit 0        # メイル用プログラムはエラーでも exit 0 すべき
elsif /.*@.*/ !~ sender then    # メイルアドレス形式でない場合
  STDERR.puts "SENDER address invalid"
  exit 0
end

to = sender
```

```
from   = rcpt
subject = NKF.nkf("-jM", 'メイル受信しました')
header = sprintf("To: %s
From: %s
Subject: %s
Content-type: text/plain; charset=iso-2022-jp
Mime-Version: 1.0\n\n", to, from, subject)
message       = NKF.nkf("-j", "ありがとう\n")
program       = sprintf("| sendmail -f %s %s", from, to)

open(program, "w") do |mail|
  mail.print header
  mail.print message
end
```

3.2.5　メイル自動応答プログラム作成時の注意

メイル配送により自動起動するプログラムはエラーで停止してはならない。デバッグする際はメイル配送時と同じ状況を作り出してプログラムを起動する。このスクリプトの例では、

- 環境変数 SENDER と RECIPIENT に値をセットする
- mbox 形式のメイルを標準入力として与える

の 2 点を満たしつつプログラムを起動する。適当なメイル格納ファイルが ~/Mail/inbox/1 にあったとすると、

```
% cat ~/Mail/inbox/1 | \
  SENDER=自分のメイルアドレス RECIPIENT=スクリプトの受信アドレス \
  ./pf3/thankyou.rb
```

のようにして確認する。

念入りに起動実験して問題がなければ、実際に電子メイル経由で起動してみる。

```
% echo test|Mail -s test-mail スクリプトの受信アドレス
```

以上のように起動しても、期待した動きが見られない場合にはシステムのメイル送信キューに溜っていないか確認する。

```
% sudo mailq
```

スクリプト起動時にエラーとなっている場合のエラーメッセージは maillog ファイルに記録されている。典型的な場所は /var/log/maillog であるのでこれを確認するとよい。

練習問題

以下の設問中 *user* は、自分のユーザ名を意味する。またメイルドメイン（@example.net）も、自身のものに置き換えて解釈せよ。また、作成したプログラムをメイル配送のタイミングで起動させるための MTA として、qmail または、dotqmail スクリプト（3.2.2 節参照）を組み込んだ Postfix システムを前提とする。

3.1 *user*-body@example.net 宛になんらかのメイルを送ると、ヘッダを取り除いた本文を YYYY-MM-DD.txt に書き込むプログラム mail-diary.rb を作成せよ。なお、YYYY-MM-DD の部分は Time.now.strftime("%F") によって得られるその日の日付である。

3.2 *user*-uranai@example.net 宛にメイルを送ると、乱数で決めた今日の運勢を送り返すプログラム mail-uranai.rb を作成せよ。送り返す宛先は環境変数 SENDER の値を利用する。完成したプログラムにより送り返されるメッセージの例を以下に示す。

```
To: user@example.net
Subject: 本日の運勢
From: user-uranai@example.net

今日の運勢は「大吉」です。
```

3.3 メイル送信による投票システム mail-vote.rb を作成せよ。たとえば、foo と bar の 2 つに投票するとして、それぞれが *user*-vote-foo と *user*-vote-bar の 2 つのメイルアドレスに対応し、有権者はどちらかに投票（送信）すると 1 票入る。ある有権者が複数の宛先に投票した場合、最後に送った宛先に投票したことになるようにする。また、このプログラムをコマンドラインから起動すると集計結果を示すようにせよ。

基本方針をある程度示す。

- プログラム起動の dotqmail ファイルを ~/.qmail-vote-default にすると、*user*-

vote-foo、*user*-vote-bar いずれの宛先も受け取り、環境変数 DEFAULT に default 文字列に相当する部分が代入される。これを利用し、ENV["DEFAULT"] の値が "foo"、"bar" いずれかだった場合のみ投票と見なす。

- 正しい候補に投票したときは有効投票、そうでないときは無効投票である旨投票者（送信者）に返信する。
- 投票数管理は、投票者のメイルアドレスをキー、その人が投票した候補者を値とするハッシュで行なう。
- コマンドライン起動かどうかの判定は、MTA によって定義されるはずの環境変数 SENDER が定義されているかで行なえばよい。

第4講

プロセスとスレッド

4.1 プロセス

1つのプログラムが単独で動くのではなく、他のプログラムを子供として起動して相互に情報のやりとりをして動かしたりすることもある。複数のプログラムを協調して動かすことにより、単独で動かすよりも効果的な処理が可能になる。そのためにはシステム上でプログラムが起動されている状態、すなわち**プロセス**についての制御が必要になる。

他のあらゆるプログラムと同様、Ruby プログラムも Unix システム上の 1 プロセスとして動いている。プロセスは新たに生成したり、生成したものに信号を送ったりなどの操作ができる。

4.1.1 プロセスの属性

Unix プロセスは 1 つのプログラムが活動している状態で、状態を示す値がいくつかある、プログラミングをする上で知っておくべき代表的なものを示す。

プロセス ID（PID）	各プロセスに振られる固有の整数
親プロセス ID（PPID）	そのプロセスを生成したプロセスのプロセス ID
ユーザ ID（UID）	プロセスの動作権限となるユーザ ID
グループ ID（GID）	プロセスの動作権限となるグループ ID
カレントディレクトリ	プロセスのその時点の作業ディレクトリ
TTY	入出力を結び付けられた端末

走行中の Ruby プログラムのシステムプロセスとしての情報は Process モジュール[注1] を介して得たり設定したりできる。以下のプログラムは起動したプロセスの PID、UID、GID を出力する。

```
#!/usr/local/bin/ruby
printf("pid=%d, UID=%d, GID=%d\n",
       Process.pid, Process.uid, Process.gid)
```

注 1　http://doc.ruby-lang.org/ja/2.1.0/class/Process.html

プロセス自身のPIDは変数$$でも参照できる。プロセスのUID、GIDは、そのプログラムを起動したユーザのUID、GIDとなる。このあと、走行中のプロセスに信号を送るシグナルについて述べるが、シグナル送信は送信者のUIDが走行中プロセスのPIDと一致しているか、スーパーユーザのID（UID=0）である場合のみ有効になる。

1 2 3 4 5 6 7 8 9 10

4.1.2　fork と exec

Unix システムは起動時に /sbin/init プロセスが起動され、残りのプロセスはすべてその子として起動される。プロセスの生成は以下の2つの機構を組み合わせて行なわれる。

1. fork　現在の自己プロセスのコピーを作り、プログラムの同じ場所から続けて動作する。
2 exec　現在の自己プロセスの情報をすべて引き継ぐ外部プログラムに実行を移す。

まず、それぞれの動きを理解するために以下の2つのプログラムを動かしてみよう。

リスト4.1●fork.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
$stdout.sync = true             # 標準出力を溜めずにすぐ書き出すため
def message(me)
  0.upto(4) do |i|
    printf("%sその%d\n", me, i+=1)
    sleep 1
  end
end

pid = fork                      # forkによってプロセスの分身発生
if pid then
  printf("こちらは親(pid=%d)\n", pid)
  message("親")
else                            # 分身側のプロセス
  sleep 0.5
  message("\t分身")
end
```

このプログラムを実行した結果は以下のようになる（pid 値はその都度異なる）。

```
% ./fork.rb
こちらは親(pid=10783)
親その1
        分身その1
親その2
        分身その2
親その3
        分身その3
親その4
        分身その4
親その5
        分身その5
```

実際に動かしてみると分かるが、1 つのプログラムの実行主体が 2 つになり、それぞれが独立して message メソッドを実行している。

続いて exec の効果を調べる。

リスト4.2●exec.rb

```
#!/usr/local/bin/ruby
exec("/bin/ls")            # ここでexecしてプロセスが完全に置き換わる
puts "Hello!!!"            # ここは実行されるか?
```

実行すると、exec 文のところで実行主体が ls コマンドになり代わり、exec より後の文が実行されずに Ruby が終了していることが分かる。

```
% ./exec.rb
cat-n-t.rb      f+e.rb          ioe-open3.rb    pipe.rb         th.rb
cat-n.rb        fork.rb         ioe-pfe.rb      popen-cal.rb    thread.html
exec.rb         i               ioe.sh          process.html    timeshock.rb
execproc.html   index.html      mpg123.rb       report.html
f+e+t.rb        intbye.rb       mutex.rb        th-op.rb
```

これら fork と exec を続けて行なうことで、1 つのプログラムから別のプログラム実行を起こ

すことが可能となる。一般的に、1つのプログラムの制御下で別のプログラムを起動するにはforkとexecを組み合わせて行なう。forkの例示プログラムにあるように、forkを呼ぶと呼び出し元となったプロセスには子プロセスのPIDが返されるが、子プロセスには`nil`が返るので、forkの返却値をもとにifで条件分岐して親プロセスの処理と子プロセスの処理を分けて記述する。

以下の例は、1つのRubyプログラムから仮想端末コマンド（例ではkterm）を起動し、それを親となるRubyプログラムから制御するものである。

リスト4.3●f+e.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
cmd = ARGV[0] || "kterm"
# コマンドライン引数に指定した場合はそれを、しない場合はktermを起動

if (pid = fork()) then
  # 親プロセスのみ必要な処理はここ
else
  # 子プロセスのみの処理はここ
  exec(cmd)
end

STDERR.printf("%s について: 終了を待つ=w killする=k 放置=その他: ", cmd)
case STDIN.gets
when /^k/i
  STDERR.puts "ボシュっ"
  Process.kill(:QUIT, pid)
when /^w/i
  STDERR.puts "じゃ、待ちます。"
  Process.wait
else
  STDERR.puts "じゃ、わしゃ勝手に終わります。さいなら。"
end
```

execは、プログラムの起動に失敗すると例外を発生させる。実際のプログラムでは外部プログラムの実行が失敗する可能性も考えなければならない。そのときの処理は4.2節「スレッド」で解説する。

4.1.3 シグナル

リスト4.3にあるProcess.killは、あるプロセスに**シグナル**を送るためのメソッドである。シグナルとはUnixシステムで走行中のプロセスに対し、「何か」が起こったことを非同期に通知するための仕組みで、どんなプロセスも走行中、自分自身に送られるシグナルを即時に受け取りそれに応じた処理に移ることも、無視することもできる。「それに応じた処理」は、シグナルを受け取ったときに自動的に呼ばれる手続きを登録することで行なわせる。自動的に呼ばれる部分を**シグナルハンドラ**という。

シグナルはシステムによって若干異なるが数十種類のものが存在する。日常的プログラミングで利用する主なものを表4.1に示す。

表4.1●シグナル

シグナル名	意味
HUP	(Hangup) 端末の回線が切断されたとき (kterm など親となる仮想端末が終了したときも該当)
INT	(Interrupt) 割り込みキー (**C-c**) をタイプしたとき
QUIT	終了キー (**C-**) をタイプしたとき
KILL	強制終了 (捕捉できない)
SEGV	(Segmantation Violation) セグメンテーション違反 (書き込み禁止メモリへの書き込みなど)
ALRM	(Alarm) 設定した制限時間が経過したときに自動的に送られる
TERM	(Terminate) kill コマンドによる強制終了 (捕捉できる)
STOP	端末以外からの実行中断 (捕捉できない)
TSTP	端末からの実行中断 (**C-z**)
CONT	中断からの復帰
USR1	ユーザ定義用 (1)
USR2	ユーザ定義用 (2)

Rubyプログラムから別のプロセスにシグナルを送るには、Process.killにシグナル名の前にコロン(:)を付けたシンボルと、シグナル送信先のプロセスIDを指定する。

なお、表中にある割り込みキーの割り当て状況はsttyコマンドで確認できる。

```
% stty -a
speed 9600 baud; 25 rows; 80 columns;
```

```
lflags: icanon isig iexten echo echoe -echok echoke -echonl echoctl
        -echoprt -altwerase -noflsh -tostop -flusho pendin -nokerninfo
        -extproc
iflags: -istrip icrnl -inlcr -igncr ixon -ixoff ixany imaxbel -ignbrk
        brkint -inpck -ignpar -parmrk
oflags: opost onlcr -ocrnl -oxtabs -onocr -onlret
cflags: cread cs8 -parenb -parodd hupcl -clocal -cstopb -crtscts -mdmbuf
        -cdtrcts
cchars: discard = ^O; dsusp = ^Y; eof = ^D; eol = <undef>;
        eol2 = <undef>; erase = ^?; intr = ^C; kill = ^U; lnext = ^V;
        min = 1; quit = ^\; reprint = ^R; start = ^Q; status = ^T;
        stop = ^S; susp = ^Z; time = 0; werase = ^W;
```

シグナル関連でないものも含まれるが、太字で示したintr、quit、suspはそれぞれINTR、QUIT、TSTPシグナルをその端末で走行中のプロセスに送るキーが、**C-c**、**C-**、**C-z**であることを示している。

4.1.4 シグナル捕捉

プログラムにシグナルが送られた場合のデフォルトの処理はシグナルごとに決まっている。たとえば、SIGINTが送られた場合はプログラムが中断させられる。これを別のものに変える場合はシグナルハンドラの登録を行なう。このためには`Signal.trap()`を用いる。

```
Signal.trap(:シグナル, ハンドラ)
```

*ハンドラ*として`nil`を指定するとシグナルを無視する。`"DEFAULT"`を指定すると独自のハンドラ登録を解除しデフォルトの処理を行なわせる。Rubyで書かれた任意の処理を行なわせたいときは`Proc`オブジェクトで、「`proc {`*処理*`}`」のように指定する。次のプログラムは**C-c**を押されてSIGINTが送られたときの処理を変えるものである。

リスト4.4●intbye.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
def bye()
  Signal.trap(:INT, nil)        # ここでのSIGINT(C-c)は無視
  STDERR.puts "そんなー。"
```

```
  sleep 1
  STDERR.puts "でもサヨナラ"
  exit 1
end

Signal.trap(:INT, proc {bye})   # SIGINTハンドラを bye メソッド呼び出しに
puts "C-cキーでは止まりません。押してみてください。"
10.downto(1) do |i|
  STDERR.printf("%d..", i)
  sleep 1
end
STDERR.puts "0 おしまい!"
```

実際に実行し、カウントダウンが進んでいる最中に **C-c** を押してみる。

```
% ./intbye.rb
C-cキーでは止まりません。押してみてください。
10..9..8..7..^Cそんなー。
でもサヨナラ
```

さらに、「そんなー」が出力されて `sleep 1` で待機している間をねらってさらに **C-c** をタイプすると無視されることが分かる。

```
10..9..8..7..^Cそんなー。
^C^C^Cでもサヨナラ
```

bye メソッド内ではすぐに SIGINT のハンドラを `nil`、すなわち無視する設定にしているため、**C-c** を連打しても止まらない。

4.1.5 シグナル捕捉の用例

実行中のプログラムは通常 **C-c** で終了する。しかし、一時ファイルを作成しているようなプログラムは **C-c** で止められたときに、一時ファイルを消去するなどの必要な後始末を行なってから exit するようにする。また、ネットワークサーバープログラムなど、端末と結び付けられずに動くプログラムでは、設定ファイルの読み直しなどの司令を端末に依らずに送ることがで

きる。USR1、USR2 シグナルはプログラム作成者が自由に定義するために用意されているシグナルで、設定ファイルの読み直しなどにしばしば利用される。

1
2
3
4
5
6
7
8
9
10

リスト4.5●sigusr1.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-

class USR1
  def initialize                # .newで呼ばれるメソッド
    @counter = 0                # 数え上げていく変数
    printf("他の端末で kill -USR1 %d\n", $$)  # $$は Process.pid と同じ
    Signal.trap(:USR1, proc {reset})
    while true
      STDERR.printf("%d..", @counter+=1)
      sleep 1
    end
  end
  def reset
    @counter = 0                # USR1シグナルが送られたらカウンタをリセット
    STDERR.puts "リセット!"
  end
end

USR1.new
```

上記プログラム実行用の端末以外に、もう 1 つ仮想端末を起動してから sigusr1.rb を実行してみる。起動直後のメッセージに出るように他の仮想端末から kill -USR1 ??? をコマンド入力すると以下のような結果が得られる。

```
% ./sigusr1.rb
他の端末で kill -USR1 15861
1..2..3..4..5..6..リセット!
1..2..リセット!
1..2..3..4..5..6..7..8..9..10..リセット!
1..2..3..4..5..^C./sigusr1.rb:11:in `sleep': Interrupt
        from ./sigusr1.rb:11:in `initialize'
        from ./sigusr1.rb:20:in `new'
```

```
        from ./sigusr1.rb:20
```

実行例は他端末で kill コマンドにて USR1 シグナルを 3 度送った後、起動した端末に戻り **C-c** をタイプして停止したものである。

4.2 スレッド

たとえば 1 人で味噌汁を作るときのことを考えよう。具材を包丁で切っているときにお湯が沸いた。人間なら包丁の手を止めてコンロを弱めるなり、出しを取る作業をするなりしてからまた包丁に戻ることができる。ところがこれに相当することをプログラミングせよとなると要領の悪い仕事になる。単純に書かれた以下のようなプログラム例で考えよう。

```
お湯を沸かし始める

While 野菜がある間
  包丁でトントン
end

沸いたお湯に入れる
```

図4.1●プログラム例

人間なら機転を利かせられるものの、コンピュータは指示以外には動けないため「包丁でトントン」の while ～ end を行なっている間は、お湯が沸騰して鍋がゆれても構うことができない。では、while から end の中に「もし、お湯が沸いたら○○する」のような if 文を入れればよいのだろうか。ところがお湯の量の割に火力が強くて、「包丁でトントン」する前に沸騰するかもしれない。となれば「もし、お湯が沸いたら○○する」を while の前の行にも入れる必要が出てくる。さらに、お湯を沸かす以外に「電子レンジで様子を見ながら冷凍食材を解凍」という処理が増えたら？　……と考えを進めると、プログラムはどんどん複雑化する。

これまで作成したプログラムでは、実行するときにまさに「実行」している部分はつねに

1 箇所である。つまり処理の流れを辿ると 1 本の線で表せる。Ruby では、動作中の単一プログラムの実行の流れを 2 本以上にすることができる。これを表現するのが Thread[注2] である。Thread を利用すると、下図の B: と C: の流れを並行して進めることができる。

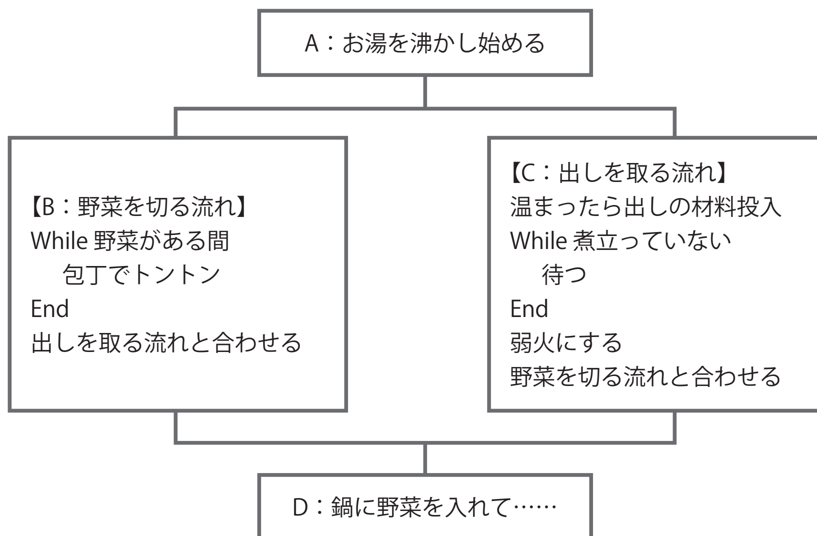


図4.2●並行処理の例

このような処理の流れのことを**スレッド**という。

4.2.1　Thread の基本的使い方

Thread 分岐した実行単位は Thread.new で作成する。

```
A
t = Thread.new do
  B
end
C
t.join
D
```

注 2　http://doc.ruby-lang.org/ja/2.1.0/class/Thread.html

このような形式で新しいスレッドが生成され、*B*の部分と*C*の部分の実行が同時に進む。joinメソッドにより、2つの処理の流れが歩調を合わせて1本に戻り、*B*、*C*両方が完了してから*D*の実行に移る。先述のforkは全く違うプロセスが2つ存在しているため、それぞれ別々の変数管理でその後のプログラム実行が進むのに対し、Threadの場合は2つの実行の流れが全く同じプロセスで動くため変数は共有される。以下のプログラムは、変数xの値を、同時進行する2つのスレッドで同時に操作している。実際に動かして試してみよ。

リスト4.6●th.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-

x = 1
t = Thread.new do
  # ここは子のみが実行するところ
  5.times do
    STDERR.print("\e[32m")      # 子スレッドが走るときに端末文字色を緑に
    x=5; sleep 0.2
  end
end
while t.alive?
  # ここが親スレッド
  STDERR.printf("\e[mx=%dに1足すと:", x) # ESC [ m で標準色に戻す
  sleep 0.05
  x += 1
  STDERR.printf("%d\n", x)
end
t.join
puts
```

このプログラムでxの値を出力しているのは太字で示した親スレッドの部分のみで、そこではそれまでのxの値に1を足した結果を出しているのにもかかわらず、出力結果にはところどころ1を足した結果になっていないものが見られる。

```
% ./th.rb
x=5に1足すと:6
```

```
x=6に1足すと:7
x=7に1足すと:8
x=8に1足すと:6
x=6に1足すと:7
x=7に1足すと:8
x=8に1足すと:6
x=6に1足すと:7
x=7に1足すと:8
  :
  :
```

1
2
3
4
5
6
7
8
9
10

なお、スレッドはいくつに分かれても、プログラムの実行主体となるプロセスは1つなので、どれか1つのスレッドがexitすると親スレッドを含めたすべてのスレッドが終了する。

4.2.2　Mutexによる競合回避

複数のスレッドで共有アクセスするデータがあるときに一連の処理を、他のスレッドから保護しつつ完遂させる必要がある場合はMutex[注3]を利用する。`th.rb`では変数xに対する処理が競合していたが、これをMutexの`synchronize`で競合から保護するように修正した例を示す。

リスト4.7●mutex.rb

```
#!/usr/local/bin/ruby
# coding: euc-jp
require 'thread'

m = Mutex.new
x = 1
t = Thread.new do
  # ここは子のみが実行するところ
  5.times do
    m.synchronize {
      STDERR.print("\e[32m")
      x=5; sleep 0.2
    }
    Thread.pass
```

注3　http://doc.ruby-lang.org/ja/2.1.0/class/Mutex.html

```
    end
  end
  while t.alive?
    m.synchronize {
      STDERR.printf("\e[mx=%dに1足すと:", x)
      sleep 0.05
      x += 1
      STDERR.printf("%d\n", x)
    }
    Thread.pass
  end
  t.join
```

実行した場合、以下のように「1 足す」という結果は正しくなる。

```
% ./mutex.rb
x=5に1足すと:6
x=5に1足すと:6
x=5に1足すと:6
x=5に1足すと:6
x=5に1足すと:6
```

synchronize ブロックの後ろにある Thread.pass は、他のスレッドに制御を譲るためのものである（後述）。

4.2.3　スレッド同士の制御

複数のスレッドを生成し、あるスレッドが別のスレッドの実行を制御するようにすると、制限時間付きのプログラムなどが比較的容易に作れる。

以下のプログラムは、次々と出る問題に対する解答の入力を 5 秒だけ待つような、制限時間付きクイズを行なうものである。

リスト4.8●timeshock.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
```

```
# 問と解を列挙。本来ファイルから読むべきだが主題はスレッドなので簡略化
questions = [
             ["この時間習っている言語は", "ruby"],
             ["ファイル一覧を出すコマンドは", "ls"],
             ["ファイルを表示・結合するコマンドは", "cat"],
             ["ファイルを削除するコマンドは", "rm"],
             ["ファイルを移動(リネーム)するコマンドは", "mv"]
  ]

n = 0                               # 問題番号
hit = 0                             # 正解数
STDOUT.sync = true
for q, a in questions
  reply = nil
  t = Thread.new do
    printf("%c[2J%c[1;1H", 27, 27) # 画面消去のエスケープシーケンス
    printf("第%d問 %s: ", n+=1, q)
    reply = gets.chomp
    if reply == a then
      print("正解!")
      hit += 1
    else
      print("ハズレ!\n")
    end
    t.exit
  end
  sleep 5
  if t.alive? then
    t.kill
    print(" 残念!\n")
    sleep 1
  end
end
printf("\n%d問正解\n", hit)
if hit < q.length/2 then
  print("トルネードです。ぐるぐるー\n")
end
```

走行中のスレッドは、別のスレッドから操作することができる。

たとえばt = Thread.new {...}としてtに得たスレッドオブジェクトを介して以下の操作ができる（抜粋）。

t.alive?	スレッドtが活きているかの真偽を返す
t.stop	スレッドtを停止させる
t.run	停止中のスレッドtの実行を再開する
t.kill	スレッドtを終了させる
t.status	スレッドtの状態を返す（"run"、"sleep"、"aborting"のいずれか）
Thread.pass	現在実行中のスレッドから他のスレッドに実行権を譲る

これらを用いて、親スレッドから子スレッドの停止・再開制御を行なう例を示す。

リスト4.9●th-op.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-

stop = false
bakudan = Thread.new do
  # 子スレッドのブロック
  puts ''                       # 1行空けておく
  begin
    timer = 11
    while true
      Thread.stop if stop       # stop変数が真ならスレッド実行停止
      # ESC 7 はカーソル位置保存、ESC [ 1 A は1行上、ESC 8 は位置復帰
      STDERR.printf("\e7\e[1A\r爆発 %2d秒前  \e8", timer-=1)
      break if timer < 1
      sleep 1
    end
  ensure
    puts "\nどかあああーーん!!" # 何をやっても必ず実行(ensureブロック)
    exit
  end
end

# メインスレッドのループ
sleep 0.01                      # こちらが僅かにあとで実行されるよう
```

```
  while bakudan.alive? do
    STDERR.print "コマンド入力(a～zのどれか):  \b"
    cmd = gets.chomp
    case cmd
    when "s"
      if stop then
        bakudan.run
        stop = false
      else
        stop = true
      end
    when "k"
      STDERR.puts "壊してみよう(Thread.kill)"
      bakudan.kill
      break
    when "q"
      STDERR.puts "逃げてみよう(exit)"
      exit
    end
    STDERR.print "\e[1A"                 # 1行上に戻しておく
  end
  puts ''
```

このプログラムを起動し、「s」を入力すると、bakudanスレッドが走行中なら停止（Thread.stop）し、停止中なら開始（bakudan.run）する。「k」を入力するとスレッドをkill（bakudan.kill）し、「q」を入力するとプログラムそのものの実行を終了する。

4.2.4　プロセスとスレッドの組み合わせ

forkとexecは外部プログラムを起動するときにしばしば利用する。すでに述べたようにexecで外部プログラムを起動することが失敗することもある。それに対応するプログラムは以下のようになる。

リスト4.10●f+e+t.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
```

```
cmd = ARGV[0] || "kterm"

if (pid = fork()) then
  # 親プロセスのみ
  th = Process.detach(pid)
  t2 = Thread.new do
    th.join
    STDERR.puts "終わったようだ"
    exit 3
  end
else
  # 子プロセスのみの処理はここ
  begin
    exec(cmd)
  rescue
    STDERR.puts "\e[31m起動失敗\e[m"
    exit 1
  end
end

STDERR.printf("%s について: 終了を待つ=w killする=k 放置=その他: ", cmd)
k = STDIN.gets

if !th.alive? then
  STDERR.puts "と、思ったらこけちゃったみたい"
  exit 2
end
case k
when /^k/i
  STDERR.puts "ボシュっ"
  Process.kill(:QUIT, pid)
when /^w/i
  STDERR.puts "じゃ、待ちます。"
  Process.wait
else
  STDERR.puts "じゃ、わしゃ勝手に終わります。さいなら。"
end
```

このプログラムは ARGV[0] を指定するとそれを起動コマンドとして exec を試みる。わざと間違ったコマンドを指定してみる。

```
% ./f+e+t.rb ktermmm
ktermmm について: 終了を待つ=w killする=k 放置=その他: 起動失敗
終わったようだ
```

Process.detach() は、引数に指定した PID を持つプロセスが終了するのを待つだけのスレッドを終了する。したがって、そのスレッドがすぐに終わったことを検出するもう 1 つのスレッドを生成しておけば、親プログラム自体を制御できるようになる。それが t2 に代入している Thread.new の部分である。

4.3 外部プログラムとの連係

世の中にはコマンドラインから利用できる便利なプログラムがたくさんある。複雑な仕事の一部を既存のプログラムに任せることで、初期の目的をこなすためのプログラムを手っ取り早く作ることもできる。

ここでは、外部プログラムを起動し、そのプロセスからの標準出力・標準エラー出力を受け取ったり、逆にそのプロセスの標準入力へデータを送り込んだりする方法をいくつか示す。

4.3.1　同期呼び出し

一つのプロセスが子プロセスを起動する場合、親となるプロセスが子プロセスの実行終了を待機してから続く処理を行なうような形態を**同期呼び出し**といい、待機せず続く処理を行なうものを**非同期呼び出し**という。同期呼び出しの場合は、子プロセスを起動して、子プロセスが出力した情報を文字列として利用する場合とそうでない場合を基準に、以下のいずれかを利用すればよい。

子プロセスの出力文字列を利用する場合

コマンド起動をバッククォートで取り込む。

例：`result = `command args``

子プロセスの出力文字列は利用しない場合

systemを利用する。

例：system "command args"

いずれの場合も、起動した子プロセスの標準出力と標準入力は親プロセスと同じ端末となる。

4.3.2 子プロセスからの出力文字列の受信

起動したプロセスが出力したものを一方的に受け取るだけであれば、子プロセスの標準出力を読み取れるような起動方法を取ればよい。その場合はIO.popenを使うのが最も簡単である。

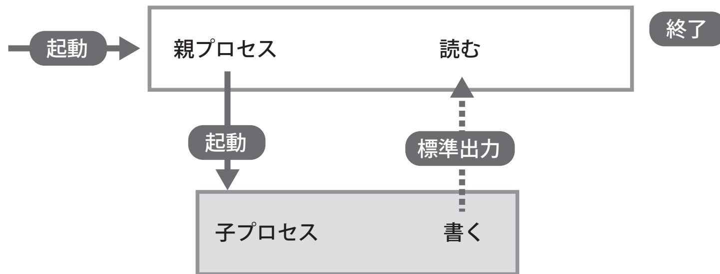


図4.3●子プロセスの標準出力を読み取る

以下のプログラムはcalコマンドを起動し、カレンダー出力を加工して出力するものである。

リスト4.11●popen-cal.rb

```
#!/usr/local/bin/ruby

lineno = 0                        # 行番号を付ける
IO.popen("cal", "r") do |c|
  while line=c.gets
    printf("%d: %s", lineno+=1, line)
  end
end
```

cal コマンドを通常起動すると以下のような結果が出力される。

```
% cal
   February 2014
 S  M Tu  W Th  F  S
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28
```

popen-cal.rb を起動すると、cal コマンドの出力を 1 行ずつ読み取って行番号を付けたものを出力する。

```
1:    February 2014
2:  S  M Tu  W Th  F  S
3:                    1
4:  2  3  4  5  6  7  8
5:  9 10 11 12 13 14 15
6: 16 17 18 19 20 21 22
7: 23 24 25 26 27 28
8:
```

このように、起動したプロセスが標準出力に書き出した文字列を順次読み取って処理するものは、ファイルの読み取りとほぼ同じ手順で作成することができる。

4.3.3　子プロセスとの双方向通信

続いて、起動したプロセスとデータの双方向のやりとりをする場合の方法を 2 つ示す。

■ 短いデータのやりとり

IO.popen でのモードを "r+" にすると、起動したプロセスとの間の読み書き両方できるようになる。

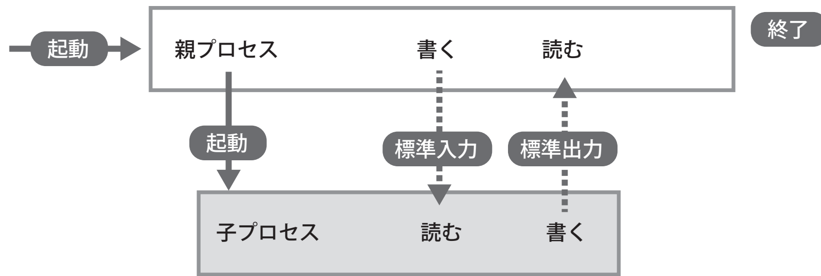


図4.4●起動したプロセスとの間の読み書き

子プロセスのやりとりが、

1. 親プロセスから子プロセスに 1 行書く
2. 親プロセスが子プロセスからの出力を読む

という短いデータの 1 往復の流れなら、

```
IO.popen(子プロセス, "r+") do |c|
  c.puts データ
  c.close_write
  while line = c.gets
    「lineを利用した処理」
  end
end
```

のように書く。close_write は出力のみをクローズするメソッドで、これがないと実際には puts をしても子プロセスまでデータが伝わらない。標準入出力経由のデータは何バイト分かを内部バッファに溜めてからまとめて書き出されるので、データを書く側が送ったつもりでも読み取り側に伝わらないことが多い。クローズすると、これ以上溜めるべきデータがないと見なされ実際に読み取り側に送られる。

■ まとまった量のやりとり

以上の書き方は比較的単純であるが、これで済む状況ばかりではない。「親→子」への出力

が複数行に渡る場合や、親と子のやりとりが何往復にも渡る場合は両プロセスでの入力処理（親となる Ruby スクリプトでは gets）が完了するタイミングが合わない。先述のバッファは有限サイズなので、書き込む側がどんどん書いて、いっこうに読まれないといずれ溢れる。溢れそうなときは書き込み自体に待ったがかかるため、処理がそこで停止する。たとえば以下のプログラムの繰り返し指定数値をいろいろ変更して試してみよ。

1
2
3
4
5
6
7
8
9
10

リスト4.12●cat-n.rb

```
#!/usr/local/bin/ruby
# ./cat-n.rb 5000 とすれば5000回。省略時1000回。

repeat = (ARGV[0] || 1000).to_i
IO.popen("cat -n", "r+") do |c|
  0.upto(repeat) {|i| c.puts i}      # cat -n コマンドに指定数値まで
  c.close_write                      # 文字列化した整数を送り続ける
  while line=c.gets                  # cat -n コマンドからの出力を得てprint
    print line
  end
end
```

繰り返し数指定を 5000 と 10000 で実行した例を示す。

```
% ./cat-n.rb 5000
     1	0
     2	1
     3	2
     4	3
(~~ 中略 ~~)
  4999	4998
  5000	4999
  5001	5000
% ./cat-n.rb 10000
(ずっと待っても何も出てこない)
```

別プロセスとの読み書きを行なう場合は、書き込みと読み込み両方をよどみなく行なう必要がある。そのためには、書き込み用の処理と読み込み用の処理を並列で動かせばよい。

スレッドを利用し、読み書きそれぞれの処理を並列で動かす例を示す。

リスト4.13●cat-n-t.rb

```
#!/usr/local/bin/ruby
# ./cat-n-t.rb 5000 とすれば5000回。省略時1000回。

repeat = (ARGV[0] || 1000).to_i
IO.popen("cat -n", "r+") do |c|
  Thread.new {                        # 別スレッドで整数文字列を
    0.upto(repeat) {|i| c.puts i}     # 一気に書き込み、
    c.close_write                     # close_write する
  }
  while line=c.gets                   # 元スレッドで読み込み
    print line
  end
end
```

繰り返し数を増やして実行してみる。

```
% ./cat-n-t.rb 10000
(～～ 省略 ～～)
  9999  9998
 10000  9999
 10001  10000
% ./cat-n-t.rb 100000
(～～ 省略 ～～)
 99999  99998
100000  99999
100001  100000
```

以上の例は、読み書きをひたすら続けるのみという単純な例だが、実際に一定の規則に従った書式のやりとりをする場合は、バッファ溢れが起きないようタイミングを合わせた入出力に注意する。

4.3.4　標準エラー出力を含めたやりとり

IO.popenでは、子プロセスからの標準エラー出力を受け取ることができない。子プロセスの標準エラー出力も受け取りたいときはRuby固有のOpen3や、C由来の言語に共通のpipe+fork+execの組み合わせを利用する。

以下に2つの方法を示すが、標準入出力・エラー出力を同時に利用する外部プログラムの例として以下のシェルスクリプトを利用する。

リスト4.14●ioe.sh

```
#!/bin/sh

echo -n "何ヶ月分?: " 1>&2
read n
if [ "$n" -le 0 ]; then          # 数字以外があるとコケる。要エラーチェック。
  echo "無効な指定です。1以上の数を指定してください。" 1>&2
  exit 1
fi
m=`date +%m`
y=`date +%Y`
while [ "$n" -gt 0 ]; do
  if [ $m -gt 12 ]; then
    m=1; y=`expr $y + 1`         # exprコマンドで変数に1を足す
  fi
  cal $m $y
  m=`expr $m + 1`
  n=`expr $n - 1`
done
```

このスクリプトは、対話的に操作して指定した月数分のカレンダーをcalコマンドに出力させるものである。入力案内のメッセージは標準エラー出力に書き出し、値の入力は標準入力から行ない、結果を（calコマンドが）標準出力に書き出している。実際に実行した例を示す。

```
% ./ioe.sh      # 次行の「何か月分?:」は標準エラー出力への出力
何ヶ月分?: 3
   February 2014
```

```
  S  M Tu  W Th  F  S
                    1
  2  3  4  5  6  7  8
  9 10 11 12 13 14 15
 16 17 18 19 20 21 22
 23 24 25 26 27 28

      March 2014
  S  M Tu  W Th  F  S
                    1
  2  3  4  5  6  7  8
  9 10 11 12 13 14 15
 16 17 18 19 20 21 22
 23 24 25 26 27 28 29
 30 31
      April 2014
  S  M Tu  W Th  F  S
        1  2  3  4  5
  6  7  8  9 10 11 12
 13 14 15 16 17 18 19
 20 21 22 23 24 25 26
 27 28 29 30
```

■ Open3 の利用

Open3 を利用すると子プロセスの標準出力、標準エラー出力、標準入力と接続したファイル記述子を生成して、それらに対して入出力を行なうことが可能となる。Open3 は、Open3::popen3 メソッドに起動するコマンド行と処理ブロックを渡す。ブロックでは標準入力、標準出力、標準エラー出力のファイルハンドルを引数として受けて処理を進める。

```
require 'open3'
Open::popen3("コマンド") do |in, out, err|
  ～～ 対話入力処理 ～～
end
```

これを用いて ioe.sh（リスト 4.14）を子プロセスで実行するプログラムを以下に示す。

リスト4.15●ioe-open3.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'open3'
prog = "./ioe.sh"

STDERR.print "指定した月数分だけカレンダーを出力します。
何ヶ月分出しますか: "
n = gets.to_i                   # 英字だけなどは0になる。
Open3.popen3(prog) do |i, o, e|
  Thread.new {                  # 子プロセスの標準入力
    i.puts n.to_s               # つまり自分(親)からの出力
    i.close
  }
  Thread.new {
    while line=e.gets           # 子プロセスの標準エラー出力
      # ここでは特に処理しない
    end
  }
  while line=o.gets             # 子プロセスの標準出力
    print line
  end
end
```

■ pipe+fork+exec の利用

親プロセスと子プロセスの間の入出力を直接作成して制御する流れを示す。Ruby 以外の言語で C 由来のものは、ほぼこれと同じ手順で子プロセスとの双方向通信ができるので覚えておくとよい。

パイプはプロセス間で通信を行なうための機構で、Ruby では、`IO.pipe` で作成する。`IO.pipe` を呼ぶと、2 つのファイル記述子を要素に持つ配列が返される。最初の（第 0）要素は読み込み用（入力端）、次の（第 1）要素は書き込み用（出力端）に利用する。出力端に書き出したデータは、同じものが同じ順で入力端から読み出せる。簡単な例で効果を示す。

リスト4.16●pipe.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-

repeat = (ARGV[0] || 5).to_i        # ./pipe.rb 10 とすると10回、省略時5回
io = IO.pipe
io[1].puts "あいうえお "*repeat      # 文字列を指定した回数繰り返したものを書き出す
io[1].flush
print io[0].gets
```

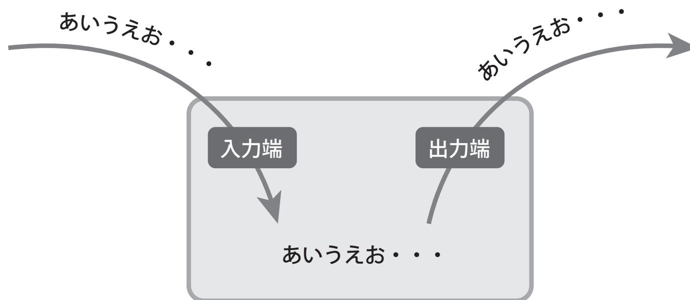


図4.5●パイプ

この例では、パイプに対してデータを書き込んですぐにそのパイプから読み込んでいるので、全く同じものが読み取れる。flush は、書き出したデータがバッファに溜められている場合に強制的に書き出すメソッドである。パイプは、fork した先のプロセスでも共有され、通常は他のプロセスとのデータの授受に利用する。標準入出力、標準エラー出力を介して子プロセスとのやりとりを行なうプログラムの流れは以下のようになる。

1. 標準入力用、標準出力用、標準エラー出力用の 3 つのパイプを作成する。
2. 標準出力をフラッシュする（1 と順不同）。
3. fork する。
   - 親プロセス
     あ．使わない記述子（パイプその 1 の入力端、パイプその 2 とその 3 の出力端）を閉じる。

い．パイプその 1 の出力端へ子プロセスへの入力を与える。

う．パイプその 2 の入力端から子プロセスの出力を得る。

- fork 先（子）プロセス

イ．標準入力をパイプその 1 入力端と置き換える。

ロ．標準出力をパイプその 2 出力端と置き換える。

ハ．標準エラー出力をパイプその 3 出力端と置き換える。

ニ．使わない記述子（パイプその 1 の出力端、パイプその 2 とその 3 入力端）を閉じる。

ホ．子プロセスを exec する。

既存のファイルハンドラを置き換えるには reopen メソッドを用いる。このような流れをプログラム化したものを次に示す。

リスト4.17●ioe-pfe.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
prog = "./ioe.sh"

STDERR.print "指定した月数分だけカレンダーを出力します。
何ヶ月分出しますか: "
n = gets.to_i                      # 英字だけなどは0になる。

i = IO.pipe                        # 子プロセスの標準入力とのパイプ
o = IO.pipe                        # 子プロセスの標準出力とのパイプ
e = IO.pipe                        # 子プロセスの標準エラー出力とのパイプ
STDOUT.flush

if pid=fork
  # ここは親プロセス
  o[1].close                       # (あ)パイプその2出力端
  e[1].close                       # (あ)パイプその3出力端
  i[1].puts n.to_s                 # (い)
  i[1].close                       # 書き出しが終了したのでクローズ
  i[0].close                       # 以下、使わないのでクローズ
  while line=o[0].gets             # (う)ioe.shからの出力を読み取る
    print ":"+line                 # コロンを付けて出力
  end
else
```

```
    # ここは子プロセス
    STDIN.reopen(i[0])             # (イ)STDINの置き換え
    STDOUT.reopen(o[1])            # (ロ)STDOUTの置き換え
    STDERR.reopen(e[1])            # (ハ)STDERRの置き換え
    i[1].close                     # (ニ)パイプその1の出力端
    o[0].close                     # (ニ)パイプその2の入力端
    e[0].close                     # (ニ)パイプその3の入力端
    exec(prog)
  end
```

このプログラムでは、独自の入力プロンプト文字列を出して値を入力し、それを起動した子プロセスの標準入力に流し込み、最後に子プロセスの標準出力を読み取り、それを加工した結果を最終出力している。その実行例を示す。

```
% ./ioe-pfe.rb
指定した月数分だけカレンダーを出力します。
何ヶ月分出しますか: 2
:   February 2014
: S  M Tu  W Th  F  S
:                   1
: 2  3  4  5  6  7  8
: 9 10 11 12 13 14 15
:16 17 18 19 20 21 22
:23 24 25 26 27 28
:
:     March 2014
: S  M Tu  W Th  F  S
:                   1
: 2  3  4  5  6  7  8
: 9 10 11 12 13 14 15
:16 17 18 19 20 21 22
:23 24 25 26 27 28 29
:30 31
```

4.3.5　子プロセス制御プログラム

■ フロントエンドプロセッサ

対話的プログラムを作る場合、ユーザインタフェースを複数用意しておけば、それだけ多くの人の要望に応えられる可能性が上がる。そのためには、実際にユーザとの対話を行なう部分と、その指示で計算機資源を制御する部分を独立したものとして作成する。制御部分は（最低）1つ作成しておき、ユーザインタフェースは簡素なものにしておく。そのまま利用させてもよいが、対話機能を受け持つ別のプログラムから制御プログラムを操作するように設計すると構造を単純化でき、見通しのよいシステムとなる。

たとえば、mpg123[注4]は、MP3ファイルを音楽信号にデコードし、音源デバイスに送り込む音楽再生プログラムであるが、このプログラム自身はコマンドラインに渡されたファイルを順次再生する機能だけを有する。一方、複数のファイルを表示してユーザに再生したいファイルを選択させたりなどのきめこまやかな対話処理を行なうプログラムを作り、そこでmpg123プログラムを子プロセスとして利用することで、利用者からは単一の音楽再生ソフトウェアに見えるものが完成する。

このように、実処理は裏で呼ぶプログラムに任せ、ユーザとの対話処理に特化したプログラムのことを**フロントエンドプロセッサ**という（逆に裏で実処理を行なうものをバックエンドプロセッサという）。

mpg123プログラムは、他のプログラムの子プロセスとして利用しやすいよう設計されている。mpg123プログラムを起動すると、標準エラー出力に現在再生中の曲の総フレーム数、再生中フレーム番号が出される。またmpg123プロセスにINTシグナルを送ると再生中の曲を停めて次の曲の再生に移る。

■ mpg123プログラム出力の解析

ここで、子プロセスとして起動し制御する対象とするmpg123プログラムが標準エラー出力に吐き出す情報を簡単に紹介しておく。実際にmpg123プログラムを起動できる場合はそれも見た方がよい。

以下の出力例は、mpg123プログラムに2つのMP3ファイルを渡し再生させ、1曲目の途中で C-c をタイプしてINTシグナルを送ったときのものである。

注4　http://www.mpg123.org/

```
% mpg123 -v music1.mp3 music2.mp3
High Performance MPEG 1.0/2.0/2.5 Audio Player for Layers 1, 2 and 3
        version 1.13.1; written and copyright by Michael Hipp and others
        free software (LGPL/GPL) without any warranty but with best wishes
Decoder: SSE

Playing MPEG stream 1 of 2: music1.mp3 ...
MPEG 1.0, Layer: III, Freq: 44100, mode: Joint-Stereo, modext: 2, BPF : 365
Channels: 2, copyright: No, original: Yes, CRC: No, emphasis: 0.
Bitrate: 112 kbit/s Extension value: 0
Frame#    74 [ 6579], Time: 00:01.93 [02:51.85], RVA:   off, Vol: 100(100)^C
[0:01] Decoding of music1.mp3 finished.

Playing MPEG stream 2 of 2: music2.mp3 ...
MPEG 1.0, Layer: III, Freq: 44100, mode: Joint-Stereo, modext: 2, BPF : 365
Channels: 2, copyright: No, original: Yes, CRC: No, emphasis: 0.
Bitrate: 112 kbit/s Extension value: 0
Frame#    53 [ 7493], Time: 00:01.38 [03:15.73], RVA:   off, Vol: 100(100)^C
[0:01] Decoding of music2.mp3 finished.
```

3段落に分かれている出力のうち、1つ目の段落はコピーライト等の表示、2つ目の段落は1曲目（music1.mp3）の再生、3つ目の段落は2曲目（music2.mp3）の再生のときのものである。着目すべきは太字で示した部分で、これらはこれから再生する音楽ファイル名と再生中のフレーム番号を示している。フレーム番号は再生が進むにつれて数が随時増えていく。mpg123プログラムを子プロセスとして起動したプログラムでは、これらの情報をパターンマッチで検出し、その後の処理に利用していくことになる。

■ mpg123制御プログラム

パイプを利用して子プロセスの標準エラー出力を読み取り、標準入力から得たユーザの司令を子プロセスにシグナル送信で伝えるプログラムの例を示す。

リスト4.18●mpg123.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-

if !ARGV[0]
```

```
  STDERR.puts "再生したいファイルを指定してください。\n用法:"
  STDERR.puts "\t#{$0} "
  exit 1
end
frame = nil                     # 生成中フレーム# の保存用
e = IO.pipe                     # mpg123からは標準エラー出力のみ読み取る

pid = fork do                   # fork&exec を行なう。PIDを記憶。
  STDERR.printf("ちわ、子です。%s を演奏します。", ARGV.join(" "))
  STDERR.reopen(e[1])
  e[0].close                    # 入力端は使わないので閉じる
  exec "mpg123", "-v", *ARGV    # *ARGVで配列を展開して渡す
end
# ここから親の処理
e[1].close                      # 親の出力端は使わないので閉じる

def prompt(file)
  STDERR.printf("\n[%s]再生中 - コマンド(nで次の曲、qで終了): ", file)
end

Thread.new do                   # 子プロセス(mpg123)と通信するスレッド
  while not e[0].closed?
    line = ""
    while l = e[0].getc         # 1字ずつ読み、lineに足していく。
      if l != "\r"[0] && l != "\n"[0]   # CRかLF以外なら
        line << l.chr           # lineに追加
      else                      # CRかLFが来たら行(line)の完成
        break                   # ループを抜けてパターンマッチに移る
      end
    end
    if /Frame\#\s*(\d+)/n =~ line
      STDERR.printf "%s ", $1 if $DEBUG
      frame = $1.to_i
    elsif /stream .* (.+) \.\.\./n =~ line
      prompt($1)          # 再生ファイル名パターンが来たらプロンプト出力
    end
  end
end

Thread.new do                   # プロセスの終了を監視するスレッド
  Process.wait                  # mpg123終了まで待機
```

```
  printf("Stopping at frame %s\n", frame)
  exit 0                        # mpg123が終了したらこのプログラムも終わる。
end

while true
  a = STDIN.gets                # promptが出ているはず
  if a.nil? || /q/ =~ a
    Process.kill(:INT, pid)     # mpg123コマンドに
    Process.kill(:INT, pid)     # 2連続でINTシグナルを送ると終了
    break
  elsif /n/ =~ a
    Process.kill(:INT, pid)     # 1回INTシグナル送信で次の曲へ
  end
end
```

練習問題

1 2 3 4 5 6 7 8 9 10

4.1 以下のように 2 つの値を入力させるだけの単純なプログラムがある。

```
STDERR.print("\e[2J")                # 画面クリア
STDERR.print "名前を入れて: "
name = gets.chomp
STDERR.print "食べたいものは: "
food = gets.chomp
```

リスト 1 行目に出力している "\e[2J" は、ESC 文字（"\e"、ASCII コード 27）のあとに [、2、J が続けて並ぶ文字列で、これは仮想端末ソフトウェアで画面をクリアするための文字並び（制御シーケンス）である。

マルチスレッドを利用し、上記のプログラムの入力中に、端末画面の左上端（1 行目の 1 桁目）に現在時刻を 0.1 秒ごとに出力し続けるプログラム clock.rb を作成せよ。画面への出力制御には以下の制御シーケンスを用いるとよい。

"\e7"	現在のカーソル位置を保存する。
"\e8"	保存したカーソル位置に戻る。
"\e[*r*;*c*H"	*r* 行、*c* 桁目にカーソルを移動する。

4.2 sigusr1.rb（リスト 4.5）を改良し、USR2 シグナルを送るとそれを捕捉し、標準エラー出力に「さようなら」と出力して終了する機能を追加したプログラム sigusr1-2.rb を作成せよ。

4.3 mpg123.rb を参考に、コマンドラインで指定したすべての引数をそのまま exec(*ARGV) で起動し、起動したコマンドの標準エラー出力文字列のみを反転赤色文字にして STDERR に出力するプログラム rederr.rb を作成せよ。たとえば、カレントディレクトリにファイル a のみがある場合に「ls -l a b」と実行すると、通常は次のような結果が得られる。

```
% ls -l a b
ls: b: No such file or directory
-rw-r--r--  1 yuuji  wheel  0 Jun 12 23:31 a
```

「./rederr.rb ls -l a b」と起動した場合に、上記実行例1行目の標準エラー出力を反転赤色文字にする。なお、反転赤色文字は

```
printf("\e[41m%s\e[m", エラーメッセージ)
```

で出力するものとし、起動コマンドがその標準エラー出力に書き出すエラーメッセージは必ず改行文字列を伴うものとしてよい（gets で読み込んでよい）。

第5講

Rubyらしい記法

長いプログラムが書けると達成感が得られるものだが、実際のところプログラムは短ければ短い程よいものである。プログラムの記述に慣れたら、すこしでも記述を短くできる記法を覚えていこう。

5.1 プログラミングを手軽にする記法

5.1.1 #{ 式 }

ダブルクォートで括られた文字列中に登場する #{ *式* } という書式は、内部の*式*を評価した値に置き換えられる。printf を持ち出すまでもない軽微なものの出力に便利である。

```
name = gets.chomp
puts "#{name}さんですね?"
```

変数だけでなく、Ruby の文として通用するどんな式でも書ける。シングルクォート中の #{ } はそのままにされる。

5.1.2 ` コマンド `

バッククォート ` ` は、内部に書かれたコマンドをシェル経由で起動し、標準出力から得られた出力を文字列として取り込む。コマンド一発で簡単に得られる情報を Ruby プログラムの中で楽に取り込みたいときに用いる。

```
systemname = `uname -s`
```

1 2 3 4 5 6 7 8 9 10

5.1.3　ヒアドキュメント

長い文字列をプログラムの中に埋め込むとき、開いたままのクォートで何行も進む代わりに、**ヒアドキュメント**注1の << を利用すると見やすいソースとなる。以下の2つのソースは同じ文字列を printf で出力する。

リスト5.1●bystr.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
title = ""
while title <= ""
  STDERR.print "タイトルは?: "
  title = gets.chomp
  if title > "" then
    printf("<html>
<head><title>%s</title></head>
<body>
<h1>%s</h1>
<p>見本文書です。</p>
</body>
</html>\n", title, title)
  end
end
```

リスト5.2●byheredoc.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
title = ""
while title <= ""
  STDERR.print "タイトルは?: "
  title = gets.chomp
  if title > "" then
    printf(<<_EOS_, title, title)
<html>
```

注1　http://doc.ruby-lang.org/ja/2.1.0/doc/spec=2fliteral.html#here

```
<head><title>%s</title></head>
<body>
<h1>%s</h1>
<p>見本文書です。</p>
</body>
</html>
_EOS_
  end
end
```

プログラム中「<<*Word*」のパターンが現れると、次の行から*Word*のみが現れる行までを「<<*Word*」の位置にある文字列として処理する。

*Word*をダブルクォートで括ったときは、置き換え文字列もダブルクォートで括られたように処理されるので、たとえば#{ }がその式の値に置き換えられる。また、*Word*をバッククォートで括ったときは、置き換え文字列をコマンドとしてシェルに渡した結果に置き換える。

5.1.4 %記法

ダブルクォートやシングルクォートを含んだ文字列や、スラッシュを含んだ正規表現を作る場合など、%記法を用いると任意の文字で内容物を括ることができる。

表5.1●%記法

記号	意味	例	結果
%Q,*string*,	" "の文字列	%Q,foo,	"foo"
%q,*string*,	' 'の文字列	%q,#{f},	'#{f}'
%x,*cmd*,	` `の文字列	%x,ls,	`ls`
%r,*regexp*,	正規表現	%r,.*/.*,	/.*\/.*/
%w,*string*,	文字列の配列（空白区切り）	%w,foo bar,	['foo', 'bar']
%W,*string*,	文字列の配列（空白区切り） #{ }展開、\有効	%W,foo #{b},	["foo", "#{b}"]

カンマ（,）の部分は任意の文字に置き換え可能。括り始めを開き括弧にした場合は、対応する閉じ括弧を右側に置く。以下のいずれも同じ意味。

```
%r,.*/.*,
%r|.*/.*|
%r(.*/.*)
%r[.*/.*]
%r{.*/.*}
%r<.*/.*>
```

HTMLのソースを吐き出すときには、出力に" "を使いたい一方、式展開をするためにダブルクォート括りの文字列にしたいことが多いので、

```
title="foo"
header=%Q,<body>
<h1 class="myclass">>#{title}</h1>
</body>,
```

と%記法を利用するとダブルクォートの記述が楽になる。

複数の単語を配列の初期値として与えるときには

```
words = %w,foo bar baz hoge hero fuga bochan,
```

のようにすると、クォート記号を書かなくていいのでソース記述が楽になる。

5.2 Enumerable、Array、Hash

Enumerableモジュール[注2]、およびArray[注3]、Hash[注4]で共通して利用できる有用なメソッドを紹介する。ArrayクラスはEnumerableモジュールをインクルードしているので、以下のメソッドはArrayに対して利用できる。また、Hashに対しては、ブロック変数を2つ与えればキーと値のペアを伴った繰り返しとして利用できる。

注2　http://doc.ruby-lang.org/ja/2.1.0/class/Enumerable.html

注3　http://doc.ruby-lang.org/ja/2.1.0/class/Array.html

注4　http://doc.ruby-lang.org/ja/2.1.0/class/Hash.html

次講で説明するファイルやディレクトリの操作では、複数のパス名が配列の形で渡される。それらの中の各要素から特定の条件に合うものを選別したり、処理を施したりするときにこれらのメソッドを効果的に適用できればプログラムソースを飛躍的に短くできる。

5.2.1 collect

各要素すべてに対してブロックを評価した値すべてを含む配列を返す。

以下の例は配列中の文字列すべてに"@example.jp"を追加し、それらを含む配列を返す。

```
user = %w,taro hanako jiro ichiro yayoi,
user.collect do |u|
  u+"@example.jp"
end
=> ["taro@example.jp", "hanako@example.jp", "jiro@example.jp",
"ichiro@example.jp", "yayoi@example.jp"]
```

5.2.2 find

先頭要素から順にブロックを評価し、真になった最初の値を返す。見付からなければnilを返す。Hashの場合は最初にブロックで真を返したキーと値のペアを要素数2の配列にして返す。

5.2.3 select

各要素に対してブロックを評価し、真を返したときの要素すべてを含む配列を返す。配列の中から特定の条件に適うものを抜き出すときに有用である。

以下の例は4つの単語を含む配列から、単語の長さが4のものを集めた配列を得ている。

```
user = %w,taro hanako jiro ichiro yayoi,
user.select {|i| i.length == 4}
=> ["taro", "jiro"]
```

5.2.4 delete

指定した値と等しい要素を削除し、削除した値を返す。以下の例は4つの単語を含む配列から、"ichiro" を削除したものである。

```
user = %w,taro hanako jiro ichiro yayoi,
 => ["taro", "hanako", "jiro", "ichiro", "yayoi"]
user.select {|i| i.length == 4}
 => ["taro", "jiro"]
```

5.2.5 delete_if、reject

ブロックを指定し、そのブロックが真を返したときの要素をすべて削除し、削除した後の配列を返す。以下の例は4つの単語を含む配列から、単語の長さが4のものを削除した配列を得ている。

```
user = %w,taro hanako jiro ichiro yayoi,
 => ["taro", "hanako", "jiro", "ichiro", "yayoi"]
user.delete_if {|i| i.length == 4}
 => ["hanako", "ichiro", "yayoi"]
```

5.2.6 grep

grep は、

```
grep(pattern)
grep(pattern) {|item| ...}
```

の書式で用い、各要素と *pattern* を === で比較した結果が真を返す要素すべてを含む配列を返す。あるファイルから特定のパターンを含む行を抽出することが grep コマンド並に手軽に実現できる。grep コマンドにパターンとファイル名を渡して該当行を出力するのとほぼ同じ挙動は

リスト5.3●grep.rb

```
#!/usr/local/bin/ruby

pattern = Regexp.new(ARGV.shift) # 第1引数はパターン
print ARGF.readlines.grep(pattern)
```

で書ける。

5.2.7 sort_by

各要素そのものをソートするのでない場合（たとえば各要素がさらに配列になっているような場合）、sort メソッドにソートブロックを指定する必要があるが、それは要素ごとの比較のたびにブロックを評価するので効率が悪い。sort_by は、あらかじめ比較要素の取り出しをまとめて行なってから並べ替えを行なうので効率がよい。以下の 2 つは同じソート規準でソートする。

```
point = [
  ["太郎", 50, 20],
  ["花子", 60, 80],
  ["二郎", 40, 50]
]

# ソートその1(sortによるブロック)
point.sort {|a, b| a[1]+a[2] <=> b[1]+b[2]}

# ソートその2(sort_byによるブロック)
point.sort_by {|a| a[1]+a[2]}
```

練習問題

5.1 YYYY/MM/DD の形式の日付文字列にマッチする正規表現を % 記法を用いて記述せよ。

5.2 氏名、メイルアドレス、金額が記載されている以下のような CSV ファイル tatekae.csv がある。

```
酒田太郎,skt@example.com,3776
鳥海二郎,chokaimt@yatex.org,2236
月山園子,gasaan@iekei.org,1984
```

これを読み込み、行の内容ごとに以下のような内容の文章を第 2 カラムに書かれた宛先と同名のテキストファイルに保存するプログラム tatekae.rb を作成せよ。なお、文章の書き込みにヒアドキュメントを利用せよ。

文書の例：

```
酒田太郎 様:
日ごろより本塾にご協力を賜りありがとうございます。

さて先日の納会にお越しいただいた際にご負担いただきました
交通費(3776円)を当方負担としてお渡ししたいと思います。
次回お越しのときに印鑑をお持ちの上、受領くださいますようお願い致します。

葛戸書房 経理課
```

5.3 1 行に 1 つメイルアドレスらしき文字列が記録されたテキストファイル、emails.txt がある。

```
foo@example.net
bar@cutt.co.jp
baz@mail.add.ress
  :
  : (以下続く)
```

このファイルを読み込み、各行のアドレスのうちドメイン部（@より後ろの部分）が存在するもののみを含んだ配列を作成するメソッド email_list を作成せよ。存在性の確認は host コマンドの返却値を整数化した $?.to_i が 0 かどうかによって判定できる。

```
res = `host www.example.com`
if $?.to_i == 0
  # 存在する
end
```

第6講

例外を意識した処理

6.1 ファイルとディレクトリ

データの処理を行なうことはコンピュータに求められる重要な仕事だが、そのデータの入れ物であるファイルの処理も同様に重要である。たとえばゲームを作る場合にハイスコアを記録したい場合を考える。データの入出力の方法も大事だが、保存するファイル名をどう決めるか、書き込みできる場所をどうやって探すか、などについても知る必要がある。

ここではファイルやディレクトリに関する操作をプログラムから行なうためのメソッドや、そのときに注意すべきシステム上の事項について説明を進める。

6.1.1 ファイルのクラスメソッド

ファイル名操作やファイルそのものの属性を操作したりするメソッドは、File クラス[注1]に集まっている。

以下のものはクラスメソッドであり、オブジェクトの確保なしにどのタイミングでも使える。

■ File.expand_path(*file* [, *dir*])

ファイル名 *file* を絶対パスに展開する。省略可能な第 2 引数 *dir* を指定すると、そのディレクトリを基準に展開する。先頭の ~ 記号はホームディレクトリに、~*user* はユーザ *user* のホームディレクトリに展開される。

```
File.expand_path(".zshrc", "~")
=> "/home/yuuji/.zshrc"
File.expand_path(".zshrc", "~/../skel")
=> "/home/skel/.zshrc"
```

シェル上で利用できる ~ 記号は、システムにとっては単なる記号であり、それをホームディレクトリに展開するのはアプリケーションの判断による処理である。

第 1 引数に絶対パス（~ で始まるものを含む）を指定した場合は、第 2 引数の如何に関らず

注 1　http://docs.ruby-lang.org/ja/2.1.0/class/File.html

第1引数を展開したものを返す。

基準ディレクトリにあるファイル名を生成したいときには、ディレクトリ名とファイル名を単に文字列として連結するのではなく File.expand_path() を用いる方がよい。

■ File.basename(*path* [, *ext*])

ファイル名 *path* のディレクトリ名を除いた部分（最後のスラッシュより後ろの部分）を返す。省略可能な第2引数 *ext* を指定すると、ファイル名の末尾がそれと一致した場合のみ *ext* をさらに削除する。

このメソッドを使用したプログラムの例を次に示す。

リスト6.1●filebn.rb

```
#!/usr/local/bin/ruby

printf("$0 = [%s].\n", $0)        # $0はこのプログラムを起動したときの名前
printf("My name is [%s].\n", File.basename($0))
printf("My root name is [%s].\n", File.basename($0, ".rb"))
```

プログラムがカレントディレクトリにある状態で起動したときと、コマンド検索パス（環境変数 PATH）に設定されたディレクトリから直接起動したときの結果を示す。

```
(カレントディレクトリに置いて起動)
% ./filebn.rb
$0 = [filebn.rb].
My name is [filebn.rb].
My root name is [filebn].
($PATHにホームディレクトリを追加してそこから起動)
% cp filebn.rb ~
% PATH=$HOME:$PATH filebn.rb
$0 = [/home/yuuji/filebn.rb].
My name is [filebn.rb].
My root name is [filebn].
% rm ~/filebn.rb
```

後者の例のように、コマンド検索パスにより起動された場合は、たとえユーザがプログラム

名しか打っていなくても、フルパス名が $0 変数に代入される。

■ File.dirname(*path*)

ファイル名 *path* のディレクトリ名部分（最後のスラッシュより前の部分）を返す。スラッシュを含まない場合は "." を返す。

File.basename と組み合わせて、動作中 Ruby プログラムの格納ディレクトリとプログラム名を得るときに利用することが多い。グローバル変数 $0 に動作中 Ruby プログラムの名前が入っていることを利用する。

```
mydir  = File.dirname($0)
myname = File.basename($0, ".rb")

configfile = File.expand_path(".#{myname}rc", "~")
scoredir   = File.expand_path("../share/#{myname}", mydir)
scorefile  = File.expand_path("#{myname}.score", scoredir)
# (不完全版でのちに修正あり)
```

一般的に、実行時に複数のファイルを利用するアプリケーションプログラムはインストールプレフィクス *prefix* を定め、

prefix/bin	実行ファイル
prefix/lib	ライブラリファイル
prefix/man	マニュアル
prefix/share	アーキテクチャ非依存ファイル（データなど）

というディレクトリ構造でインストールすることが多い。プログラムの利用者がそのプログラムをどの *prefix* にインストールするかは自由なので、プログラムでは自分で自分の *prefix* を検出し、データが格納されるべき適切なディレクトリを探し出す必要がある。

■ File.chmod(*mode*, *file*)

file の属性を *mode* に変更する。失敗したときは例外が発生する。ファイルの属性は 8 進数で最大 4 桁の値で指定する。

表6.1●ファイルの属性

特殊属性			所有者			グループ			その他		
4	2	1	4	2	1	4	2	1	4	2	1
setuid	setgid	sticky	r	w	x	r	w	x	r	w	x

各桁の意味は以下のとおり。

setuid　プログラム実行時に、プログラム自身のファイル所有者の user id でプロセスを起動する。

setgid　プログラム実行時に、プログラム自身のファイル所有者の group id でプロセスを起動する。

sticky　対象がディレクトリの場合、そのディレクトリに書き込み権限を持つユーザであっても、他者の所有するファイルを消すことはできない。

r（Readable）　そのファイルに対して、所有者／同一グループ所属ユーザ／それ以外のユーザが読み出し操作を行なえるか。

w（Writable）　そのファイルに対して、所有者／同一グループ所属ユーザ／それ以外のユーザが書き込み操作を行なえるか。

x（eXecutable）　そのファイルに対して、所有者／同一グループ所属ユーザ／それ以外のユーザが実行操作を行なえるか。ただし、対象がディレクトリの場合は chdir できるか。

Ruby では 0（ゼロ）で始まる数値は 8 進数指定となる。たとえば、

```
File.chmod(0754, "newprog.rb")
```

とすると、newprog.rb というファイルに対し、所有者は、読み・書き・実行すべて、同一グループユーザは読みと実行、その他のユーザは読み取りのみできるように属性設定する。

なお、ファイルそのものの作成や、ファイル名の変更はそのファイルが置かれるディレクトリの属性が可否を決める。

File.unlink(*filename*)

ファイルを削除する。厳密には、ファイルを消すわけではなく、ファイル名 *filename* の、実体との結合を削除する。

Unix ファイルシステムではファイルの実体に結び付いている名前を複数持てる。ファイルの新規作成では 1 つ目の名前が付き、2 つ目以降は ln コマンドで作れる。

以下のように続けてコマンド起動した流れを説明する。

```
% echo Hello > hoge
% ln hoge hero
% ln hero heso
% echo world > hoge
% rm hoge
```

まず、文字列 Hello という内容で hoge ファイルを作成する。

```
% echo Hello > hoge
% ls -l hoge
-rw-r--r--  1 yuuji  wheel  6 Mar 11 13:30 hoge
(第2フィールドの 1 はリンクカウント)
```

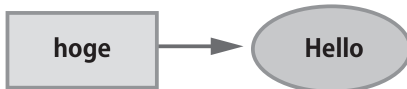


図6.1●リンクの最初の状態

ファイルシステムの中で "Hello" という内容を持つ領域が確保され、そこを指すように hoge という名前がディレクトリに作成される。続いて、ln コマンドを用いて同一ファイルの内容に別のリンクを作成する。

```
% ln hoge hero
% ls -l h*
-rw-r--r--  2 yuuji  wheel  6 Mar 11 13:30 hero
-rw-r--r--  2 yuuji  wheel  6 Mar 11 13:30 hoge
% ls -li h*
```

```
1510107 -rw-r--r--  2 yuuji  wheel  6 Mar 11 13:30 hero
1510107 -rw-r--r--  2 yuuji  wheel  6 Mar 11 13:30 hoge
(ls -i オプションでinode番号も出力)
```

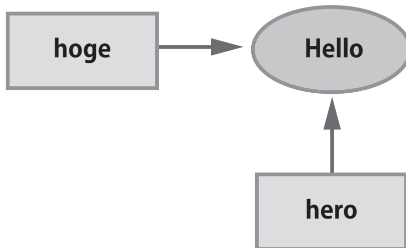


図6.2●リンクを1つ増やした状態

hoge ファイルの指す内容と同一のものを指すように、hero という名前が同じディレクトリに作成される。上の例で ls コマンドに指定した -i オプションはファイルの内容の inode 番号も出力する。inode 番号は同一ファイルシステムで唯一となるように割り当てられる番号である。2 つのファイル hoge、hero ともに 1510107 番で示される実体を指していることが分かる。さらに続けて 3 つ目のリンク heso も作成してみる。

```
% ln hero heso
% ls -li h*
1510107 -rw-r--r--  3 yuuji  wheel  6 Mar 11 13:30 hero
1510107 -rw-r--r--  3 yuuji  wheel  6 Mar 11 13:30 heso
1510107 -rw-r--r--  3 yuuji  wheel  6 Mar 11 13:30 hoge
(リンクカウントが3になる)
```

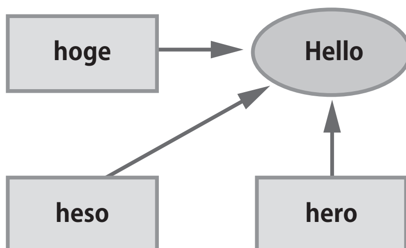


図6.3●リンクをさらに1つ増やした状態

3 つのファイルがすべて inode 1510107 の実体を指している。

ファイルの中味を修正してみる。"Hello" だった内容を "world" に変更する。

```
% echo world > hoge
% ls -li h*
1510107 -rw-r--r--  3 yuuji  wheel  6 11 Mar 13:32 hero
1510107 -rw-r--r--  3 yuuji  wheel  6 11 Mar 13:32 heso
1510107 -rw-r--r--  3 yuuji  wheel  6 11 Mar 13:32 hoge
(すべて変わる(タイムスタンプに注目))
% rm hoge       (hogeという名前だけを消す)
% ls -li h*
1510107 -rw-r--r--  2 yuuji  wheel  6 11 Mar 13:32 hero
1510107 -rw-r--r--  2 yuuji  wheel  6 11 Mar 13:32 heso
```

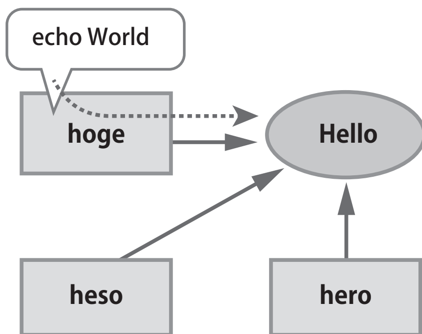


図6.4●リンクをたどってファイルの内容を変更

一般的に「ファイルを消す」という行為は、ファイルへの 1 リンクを消しているに過ぎない。リンクカウント 1 のときに unlink するとファイルを参照できなくなる、つまり実質的に「削除される」が、実体のデータがディスク上から消えたわけではない。

ちなみに rm コマンドはファイルへのリンクを消すだけであるが、BSD 系 OS の rm[注 2] にある -P オプション、Linux 系 OS の shred コマンド[注 3] はファイルの中味を上書きすることを試みる。これらを用いると、伝統的なファイルシステム上のファイルであればデータをディスク上から抹消できる。

注 2　http://www.jp.freebsd.org/cgi/mroff.cgi?sect=1&cmd=&lc=1&subdir=man&dir=jpman-6.0.0%2Fman&man=rm

注 3　http://linuxjm.sourceforge.jp/html/GNU_coreutils/man1/shred.1.html

■ File.link(*old*, *new*)

old が指すファイルを指す新しいリンク *new* を作成する。同一のファイルに直接結び付けるリンクを**ハードリンク**という。上記実行例の ln コマンドによって作られているリンクがハードリンクである。

■ File.symlink(*old*, *new*)

old という名前を指す *new* というシンボリックリンクを作成する。ln コマンドの -s オプションでもシンボリックリンクは作成できる。ハードリンクに対するものとして、シンボリックリンクのことを**ソフトリンク**ということもある。

```
% ln -s foo bar
% ls -lF bar
lrwxr-xr-x  1 yuuji  wheel  3 Jun  8 07:23 bar@ -> foo
```

bar というファイル名へのアクセスは foo へのアクセスに変換される。もちろん foo がなければアクセスは失敗する。

```
% ls foo
ls: foo: No such file or directory
% cat bar
cat: bar: No such file or directory
% date > foo
% cat bar
Mon Jun  8 07:26:34 JST 2014
(fooがあればbarへのアクセスも成功)
```

■ File.rename(*from*, *to*)

from というファイル名を *to* に変える。*to* という名前で別のファイルが存在するときは上書きされる。失敗すると例外が発生する。

■ File.stat(*filename*)

filename の情報を取得する。File::Stat オブジェクト[注4] が返る。

注 4　http://doc.ruby-lang.org/ja/2.1.0/class/File=3a=3aStat.html

6.1.2　ファイル入出力時の特殊なメソッド

単一ファイルを何度も読み書きする場合は、その都度閉じたり開いたりを繰り返すのではなく、対象ファイルのうち次に読み書きする位置（ファイルポインタ）を移動したり、書き込んだ内容を確実にディスクに書き出す処理が必要になる。以下のメソッドは IO クラス[注5]の非クラスメソッドである。すでに open している IO オブジェクトから利用する。

■ seek(*offset* [, *whence*])

ファイルポインタを *offset* だけ移動する。省略可能な第 2 引数で移動の基準位置を指定する。

IO::SEEK_SET	ファイルの先頭（デフォルト値）
IO::SEEK_CUR	現在のファイルポインタ位置
IO::SEEK_END	ファイルの末尾

■ rewind

ファイルポインタを先頭に移動する。

■ pos、tell

現在のファイルポインタ位置を返す。

以下の例はファイル読み取りが末尾まで達するたびに、きまぐれ（random）な位置にファイルポインタを変えて何度もファイルを読むプログラムである。

リスト6.2●randseek.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-

thisfile = "randseek.rb"        # このファイルの名前
sz = File.size(thisfile)
open(thisfile, "r") do |file|
```

注 5　http://doc.ruby-lang.org/ja/2.1.0/class/IO.html

```
    while true
      while line = file.gets
        print line
      end
      puts "読み終わりました。[Enter]でもう一度。やめたい場合は C-c"
      gets
      file.seek(rand(sz))         # 乱数でファイルポインタを移動
      printf("%dバイト目から読み直します。\n", file.pos)
    end
  end
```


■ flush

IO ポートの内部バッファをフラッシュする。STDOUT のようにバッファリングされる IO ポートへの出力をフラッシュすることでその時点までのデータが書き込まれる[注6]。

■ sync

出力同期モードを真偽値で指定する。

```
IO.sync=true
```

なら同期モード、

```
IO.sync=false
```

とすると非同期（バッファリングされる）モードになる。

■ 出力のバッファリング

プログラムから出力を行なう場合、print などの書き込み操作をしてもすぐに実際に対象デバイスにデータが書き込まれるとは限らない。効率を上げるため一定量を溜めてからまとめて書き出す処理が行なわれる。このため、目の前の短期的な応答性が求められるプログラムでは、出力先のバッファリングを制御する必要がある。

注 6　fsync（http://doc.ruby-lang.org/ja/2.1.0/class/IO.html#I_FSYNC）が必要な場合もある。

バッファリングの有無による違いを確認するために、Ruby 1.8 での挙動を用いた例を示す。システムの標準的な設定として、標準出力はデフォルトでバッファリングされ、標準エラー出力はデフォルトでバッファリングされない。他の言語処理系を使う場合にもこのような注意が必要な点は同じである。

Ruby 1.8 用の以下のプログラムは、標準出力から "O" を、標準エラー出力から "E" を 1 字ずつ交互に出力する。

リスト6.3●oeoe.rb

```
#!/usr/local/bin/ruby18

3.times do              # 全体を3回繰り返す
  12.times do           # STDOUT/STDERR 交互出力を12回繰り返す
    STDOUT.print "O"
    STDERR.print "E"
  end
  STDOUT.puts ""        # 12回終わったらSTDOUTに改行出力
end
STDOUT.puts "Done."
```

このプログラムでは、交互出力を 12 回行なったあと標準出力に改行文字を出力している。NetBSD6/amd64 で実行すると以下のようになった。

```
% oeoe.rb
EEEEEEEEEEEEOOOOOOOOOOOO
EEEEEEEEEEEEOOOOOOOOOOOO
EEEEEEEEEEEEOOOOOOOOOOOO
Done.
```

この例から、標準出力は改行文字までがまとめて出力されていることが分かる。プロセスと通信を行なう場合などで、出力先にデータを即時に送りたい場合は flush メソッドで書き出しバッファを一掃するか、常に即時送りをした場合は sync=true で出力同期モードに変える。flush を用いるように変えたプログラムと実行例を示す。

リスト6.4●oeflush.rb

```
#!/usr/local/bin/ruby18

3.times do               # 全体を3回繰り返す
  12.times do            # STDOUT/STDERR 交互出力を12回繰り返す
    STDOUT.print "O"
    STDERR.print "E"
    STDOUT.flush
  end
  STDOUT.puts ""         # 12回終わったらSTDOUTに改行出力
end
STDOUT.puts "Done."
```

```
% oeflush.rb
EOEOEOEOEOEOEOEOEOEOEOEO
EOEOEOEOEOEOEOEOEOEOEOEO
EOEOEOEOEOEOEOEOEOEOEOEO
Done.
```

ただし、即時書き込みはプログラムの動作性能を落とす可能性があることや、実際に書き出されるかどうかは受け取り側の状況によることに注意する必要がある。

6.1.3　ディレクトリ

ファイルはディレクトリに格納する。保存ファイルを書き込める場所を探したり、既存のファイルを特定のディレクトリから探したりするためには、ディレクトリを操作するためのメソッドの集まった Dir クラス[注7]を利用する。

■ `Dir[pattern]`、`Dir.glob(pattern)`

pattern にシェルパターンとしてマッチするファイル名一覧を含む配列を返す。

パターンの一部に利用して特別な意味を持つ記号は以下のとおり。

注7　http://doc.ruby-lang.org/ja/2.1.0/class/Dir.html

表6.2●パターン指定に利用する特殊記号

記号	説明
*	0文字以上の任意の文字列にマッチする。
?	任意の1字とマッチする。
[ ]	括弧内に列挙した任意の1字とマッチする。ハイフンで繋ぐとその範囲の文字のどれか（例：0-9)、括弧内の先頭が^のときは指定以外の文字とマッチする。
{ }	組み合わせ展開。例：file.{c,h,rb} は file.c file.h file.rb に展開される。
**/	ディレクトリ再帰指定。**/foo とするとカレントディレクトリ以下すべてのディレクトリを対象に foo を探す。

■ Dir.entries(*dir*)

ディレクトリ *dir* の持つファイルエントリを配列として返す。*dir* ディレクトリを基準としたファイル名なのでアクセスする場合は *dir* に Dir.chdir するか、File.expand_path[注8] で絶対パスに展開する必要がある。

すべてのディレクトリのファイルエントリは［".", ".."］で始まる。irb を起動した上で、Dir.entries(".")を確認してみる。

```
Dir.entries(".")
=> [".", "..", "direntries.rb", "dirfile.html", "exception.html"]
```

得られたリストを元に、各ファイルに関する情報を表示するプログラムを示す。File::Stat の詳細は次項で説明する。

リスト6.5●direntries.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
checkdir = ARGV.shift || "."
printf("(1)ここは%s\n", Dir.pwd)
printf(%Q/"%s" ディレクトリのエントリ一覧\n/, checkdir)
files = Dir.entries(checkdir)

printf(%Q/先頭要素は必ず  "%s"\n/, files.shift)
```

注8　File.expand_path（http://doc.ruby-lang.org/ja/2.1.0/class/File.html#S_EXPAND_PATH）
File.stat（http://doc.ruby-lang.org/ja/2.1.0/class/File.html#S_SIZE）

```
printf(%Q/2番目要素は必ず "%s"\n/, files.shift)
print("残りは登録順: ")
$KCODE=ENV["LC_ALL"] || ENV["LANG"] || 'u' # for 1.8
p files

puts "-"*15+"サイズ一覧"+"-"*15
files.each do |f|
  file = File.expand_path(f, checkdir)
  size = File.stat(file).size
  printf("%-30s %6d\n", f, size)
end
```


■ **Dir.pwd、Dir.getwd**

現行 Ruby プロセスのカレントディレクトリのフルパスを返す。

■ **Dir.chdir(*dir*)**

カレントディレクトリを *dir* に変更する。後ろにブロックを指定するとブロック実行中のみカレントディレクトリを変更する。

リスト6.6●pwd.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
printf("(1)ここは%s\n", Dir.pwd)
Dir.chdir(File.expand_path("~")) do
  printf("(2)ここは%s\n", Dir.pwd)
end
printf("(3)ここは%s\n", Dir.pwd)
```

実行すると以下のようになる。

```
% ./pwd.rb
(1)ここは/home/yuuji/Ruby
```

```
(2)ここは/home/yuuji
(3)ここは/home/yuuji/Ruby
```

■ Dir.mkdir(*newdir* [, *mode*])

新しいディレクトリ *newdir* を作成する。省略可能な第 2 引数 *mode* を指定すると、その時点の umask 値でマスクした値の属性値となる。umask とは、新規にファイルやディレクトリを作成する場合に、ファイル属性のどのビットを落とす（0 にする）かを決める値である。一般的には umask を 022（8 進値）にしておいて、「グループ」と「その他」の 2 のビット（書き込み権）を出さないのをデフォルトのファイル属性とする（123 ページ、表 6.1 参照）。Ruby では umask 値を File.umask[注9] が保持している。したがって、Dir.mkdir に *mode* を指定した場合、実際に作成されるディレクトリの属性値は *mode* & ~File.umask となる。

■ Dir.rmdir(*dir*)

ディレクトリ *dir* を削除する。成功すると 0 が返され、失敗すると例外が発生する。

6.1.4 ファイルの消えるタイミング

ファイルはリンクカウント 1 のときの名前が消えても、そのファイルへのアクセスがある限りファイルシステムの領域を占め続ける。プログラム中で利用する一時ファイルを作成後、すぐに unlink しても close するまではアクセスし続けられる。

リスト6.7●unlinktest.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
tmpfile = ARGV.shift || "00TEMPFILE"
tmpdir  = File.dirname(tmpfile)

STDERR.puts "最初のこのディレクトリの使用量:"
system "du -sk #{tmpdir}"
open(tmpfile, "w+") do |tf|
```

注 9　http://docs.ruby-lang.org/ja/2.1.0/class/File.html#S_UMASK

```
  1.upto(10) do |lineno|
    tf.printf("これは%02d行目\n", lineno)
  end
  tf.puts "-"*1024*1024         # 1MBのダミーデータ
  tf.flush                      # バッファを書き込み
  tf.rewind                     # ファイルポインタを先頭に
  system "ls -lF #{tmpfile}"
  STDERR.puts "ファイル作成直後のこのディレクトリの使用量:"
  system "du -sk #{tmpdir}"
  STDERR.puts "改行を押すとこのファイルを消します。"
  STDIN.gets
  File.unlink(tmpfile)
  system "ls -lF #{tmpfile}"
  STDERR.puts "closeする前のこのディレクトリの使用量:"
  system "du -sk #{tmpdir}"
  STDERR.puts "1行目から読みます。"
  STDERR.puts "別の端末でlsして#{tmpfile}がないことを確認しましょう。"
  10.times do |l|
    print tf.gets
    sleep 1
  end
end
STDERR.puts "closeされたあとのこのディレクトリの使用量:"
system "du -sk #{tmpdir}"
```


このプログラムを実行すると以下の結果が得られる。

```
% ./unlinktest.rb
最初のこのディレクトリの使用量:
116     .
-rw-r--r--  1 yuuji  wheel  1048757 Mar 13 16:00 00TEMPFILE
ファイル作成直後のこのディレクトリの使用量:
1156    .
改行を押すとこのファイルを消します。

ls: 00TEMPFILE: No such file or directory
closeする前のこのディレクトリの使用量:
1156    .
1行目から読みます。
```

```
別の端末でlsして00TEMPFILEがないことを確認しましょう。
これは01行目
これは02行目
これは03行目
これは04行目
これは05行目
これは06行目
これは07行目
これは08行目
これは09行目
これは10行目
closeされたあとのこのディレクトリの使用量:
116     .
```

4 つの数字が出されているタイミングは、

1. データファイル作成前（116）
2. データファイル作成直後（1156）
3. データファイル削除後で open している間（1156）
4. データファイル作成後で close したあと（116）

のとおりで、3. のときにはディレクトリエントリからデータファイルの名前がなくなっているにも関らず、ディスク上に確保された領域が残っている状態であることが分かる。

ユーザに見せる必要がなく、プログラム実行終了に不要となるファイルは open 直後に unlink するとよい。

6.2 ファイルテスト演算子

ファイルへの書き込み処理などは予定どおり完了するとは限らない。実際には書き込み権限がない場合などがあるため、事前にファイルに関連する条件を調査することが重要である。ファイルが存在するか、読み書き可能かなどといったファイルの状態を調べるための組み込み

関数 test[注10] を利用して、これらの条件判定を行なう。

1 2 3 4 5 6 7 8 9 10

6.2.1　test の用法

test は次の書式で用いる。

```
test(cmd, file1 [, file2])
```

指定したファイル *file*$_1$ が *cmd* の要件を満たすかどうかを返す。2 つのファイルの関係を調べるための *cmd* には第 3 引数 *file*$_2$ を与える。

表6.3●testのcmd

cmd	働き
?r	現在の実行 uid で読めるか
?w	現在の実行 uid で書き込めるか
?x	現在の実行 uid で実行できるか
?o	現在の実行 uid が所有するファイルか
?G	現在の実行 gid と同じグループ所有か
?R	実 uid で読めるか
?W	実 uid で書き込めるか
?X	実 uid で実行できるか
?O	実 uid が所有するファイルか
?e	存在するか
?s	0 バイトでないか（サイズが返る）
?f	普通のファイルか
?d	ディレクトリか
?l	シンボリックリンクか
?p	名前付きパイプ（FIFO）か
?b	ブロックデバイスファイルか
?c	キャラクタデバイスファイルか
?u	setuid ビットがセットされているか
?g	setgid ビットがセットされているか

注 10　http://doc.ruby-lang.org/ja/2.1.0/class/Kernel.html#M_TEST

cmd	働き
?k	sticky ビットがセットされているか
?M	最終更新時刻を返す
?A	最終アクセス時刻を返す
?C	inode 時刻を返す
以下は第 2、第 3 引数間の比較	
?-	両ファイルが同一か
?=	両ファイルの最終更新時刻が等しいか
?>	file_1 の最終更新時刻が file_2 より大きい（新しい）
?<	file_1 の最終更新時刻が file_2 より小さい（古い）

6.2.2　test の利用例

実行結果を決まった名前のファイルに保存するようなコマンドを考える。ここでは単純なものとしてたとえば、おみくじの結果をカレントディレクトリの fortune.txt に保存するプログラムを考える。

リスト6.8●ft0.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
#Foretune teller - version 0
ftune  = %w,大吉 吉 中吉 小吉 半吉 末吉 小凶 凶,
outfile       = "fortune.txt"
srand
result = ftune[rand(ftune.length)]
open(outfile, "w") do |f|
  f.printf("今日の運勢は %s\n", result)
end
STDERR.puts "#{outfile}を見よ。"
```

起動して、fortune.txt ファイルを確認してみる。

```
% ./ft0.rb
```

```
fortune.txtを見よ。
% ls -l fortune.txt
-rw-r--r--  1 yuuji  wheel  26 Mar 14 07:58 fortune.txt
% cat fortune.txt
今日の運勢は 中吉
```

もう一度実行すると上書きされる。

```
% ./ft0.rb
fortune.txtを見よ。
% cat fortune.txt
今日の運勢は 大吉
```

せっかく大吉が出たので上書きされないよう chmod で書き込み禁止にする。

```
% chmod -w fortune.txt
% ls -l fortune.txt
-r--r--r--  1 yuuji  wheel  26 Mar 14 07:58 fortune.txt
```

この状態で実行するとプログラムはエラー終了する。

```
% ./ft0.rb
ft0.rb:8:in `initialize': Permission denied @ rb_sysopen - fortune.txt (Errno::EACCES)
        from ft0.rb:8:in `open'
        from ft0.rb:8:in `<main>'
```

以上の様子から、次のような問題があることが分かる。

- 1 回前の結果が消える。
- 結果ファイルに書き込みできないときにエラーが発生する。

これらの問題を解決するなら以下のような改良を施すことになるだろう。

- 結果書き込み前にバックアップファイルに移動しておく。
- 書き込みできるか事前に確かめる。

6.2.3 すべてのファイルに対しての処理

あるディレクトリ以下にあるすべてのファイルに対して特定の処理を行なうことはしばしば必要となる。そのための手順は以下のようになる。

1. あるディレクトリを開き、その中のすべてのファイルに対して以下の処理を行なう。
2. それがファイルなら特定の処理を行なう。
3. それがディレクトリならそのディレクトリに対してこの処理全体を行なう。

特定の処理をメソッド化し、処理対象がディレクトリなら再帰処理を行なう。以下のプログラムは、指定したディレクトリ（省略時はカレントディレクトリ）以下すべてのファイルに対して、それが通常ファイルならバックアップファイルを作成する。

リスト6.9●mkbak-r.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
dir = ARGV.shift || "."

def cp(from, to, bufsize=8192)
  open(from, "r") do |src|
    open(to, "w") do |dest|
      while s = src.read(bufsize)
        dest.write s
      end
    end
  end
end

def bakdir(dir)
  ext = ".bak"
  Dir.foreach(dir) do |f|
    file = File.expand_path(f, dir)
    case f
    when ".", ".."              # カレントディレクトリと親ディレクトリ
      next                      # は飛ばす
    when %r,#{ext}$,            # それが既にバックアップファイルなら
      next                      # やはり飛ばす
```

```
    else
      if test(?d, file)         # ディレクトリなら
        bakdir(file)            # 再帰的に自分を呼ぶ
      elsif test(?f, file)      # 通常ファイルなら
        # バックアップファイル作成
        bak = File.expand_path(f+".bak", dir)
        File.unlink(bak) if test(?e, bak)
        cp(file, bak)
      end
    end
  end
end

bakdir(dir)
```

1 2 3 4 5 6 7 8 9 10

6.3 例外処理

プログラムの不具合にはさまざまな段階がある。文法的なエラーなどプログラムの実行そのものができない単純なもの、実行はできるが問題解決法が違うなど論理的な誤りのものについては、いずれもプログラムを作成している場所で判断して修正することができる。ところがそれらの問題がないプログラムでも、実際に実行するときの外界条件によって予期した動きを取れなくなるような不具合がある。たとえば、データをファイルに保存しようとしたのにディスク容量不足などが原因で書き込みができないような状況ではプログラムは目的を達成できない。

このように、特定の条件によって想定した処理を進められないような事態のことを**例外**という。プログラム実行時にそうした事態が生じたときには例外を意味するException[注11]が発生される。

注11 http://doc.ruby-lang.org/ja/2.1.0/class/Exception.html

6.3.1 例外発生

文法的にはエラーのない以下のプログラムを動かしてみよ。

リスト6.10●openerr.rb

```
#!/usr/local/bin/ruby

file = ARGV.shift || "nonexistent.txt"
open(file, "r") do |f|
  while line = f.gets
    print line
  end
end
```

実行すると例外が発生する。

```
openerr.rb:4:in `initialize': No such file or directory @ rb_sysopen -
 nonexistent.txt (Errno::ENOENT)
        from openerr.rb:4:in `open'
        from openerr.rb:4:in `<main>'
```

6.3.2 begin 〜 rescue 〜 end

おみくじプログラム（リスト 6.8）の例に戻ると、改良すべき点として次の 2 つが挙げられる。

- 結果書き込み前にバックアップファイルに移動しておく。
- 書き込みできるか事前に確かめる。

ただし、これらにはそれぞれ落とし穴がある。

既存の結果ファイル fortune.txt をバックアップファイル fortune.bak に移動できるかどうかは、どうすれば確かめられるだろう。場合によってはバックアップファイルである fortune.bak の存在やその書き込み可能性について確かめたくなるが、実際にはこれは「バックアップファイルを作れるかどうか」には関係ない。以下の実験で、移動先のファイルを書き

込み禁止にして移動を行なってみる。

```
% mkdir dirtest
% cd dirtest
% echo This is foo > foo
% echo This is backup file > foo.bak     (foo.bakがバックアップファイル)
% chmod -w foo.bak                       (書き込み禁止にする)
% ls -lF
total 1
-rw-r--r--  1 yuuji  wheel  12 Mar 15 12:59 foo
-r--r--r--  1 yuuji  wheel  15 Mar 15 12:59 foo.bak
% echo hoge > foo.bak
zsh: permission denied: foo.bak
(直接ファイルに書こうとするとエラーになるが...)
% mv foo foo.bak
override r--r--r--  yuuji/wheel for foo.bak? n
(mvの場合は警告が出るがnと答える)
% irb
irb> File.rename("foo", "foo.bak")
=> 0
irb> exit
% ls -lF
total 1
-rw-r--r--  1 yuuji  wheel  12 Mar 15 12:59 foo.bak
(12バイトなので元々fooだったものがfoo.bakになっている)
```

ファイルを rename する場合は、移動先ファイルを修正するのではないので移動先ファイルのパーミッションは関係ない。元のファイルを消してそのファイルを作る権利があるかが意味を持つので、ディレクトリに書き込み権があるかが意味を持つ。

```
% touch foo foo.bak
% chmod -w .               (ディレクトリを書き込み禁止にする)
% irb
irb> File.rename("foo", "foo.bak")
Errno::EACCES: Permission denied @ sys_fail2 - (foo, foo.bak)
        from (irb):1:in `rename'
        from (irb):1
        from /usr/local/bin/irb21:11:in `<main>'
```

これは、ファイルを新規作成するときも同様の要件となる。

以上のことから、前掲の2点の改良についてはディレクトリへの書き込み権限を検査する必要があることが分かる。

リスト6.11●ft1.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
#Foretune teller - version 1
ftune  = %w,大吉 吉 中吉 小吉 半吉 末吉 小凶 凶,
outfile       = "fortune.txt"
srand
result = ftune[rand(ftune.length)]

if test(?w, ".") then           # ディレクトリの書き込み権限を確認
  if test(?s, outfile) then
    ext = File.extname(outfile)
    bak = File.basename(outfile, ext)+".bak"
    File.rename(outfile, bak)
  end
  open(outfile, "w") do |f|
    f.printf("今日の運勢は %s\n", result)
  end
  STDERR.puts "#{outfile}を見よ。"
end
```

しかしこれでも不十分で、ファイルシステムが溢れるなど、書き込みに失敗する要因は他にいくらでもある。だからといって、それらに対する事前予想を if で書き連ねてもプログラムの本筋が見づらくなる。

このような場合には、begin ～ rescue ～ end 構文を用いる。エラーが起きない理想的な場合の処理のみをまず書き、エラーが起きた場合の処理をまとめて別箇所に書くことで、処理の流れがはっきりし、エラーが起きた場合の対処ももれなく書くことが容易になる。

リスト6.12●ft2.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
```

```
#Foretune teller - version 2
ftune  = %w,大吉 吉 中吉 小吉 半吉 末吉 小凶 凶,
outfile       = "fortune.txt"
srand
result = ftune[rand(ftune.length)]

begin
  if test(?s, outfile) then
    ext = File.extname(outfile)
    bak = File.basename(outfile, ext)+".bak"
   # File.rename(outfile, bak)
  end
  open(outfile, "w") do |f|
    f.printf("今日の運勢は %s\n", result)
  end
  STDERR.puts "#{outfile}を見よ。"
rescue
  abort(<<MSG)
書き込みに失敗しました。ディレクトリへの書き込み権限を確認してください。
MSG
end
```

この場合は例外発生時に復旧せず、異常終了している。そのためのメソッドが abort である。

6.4 一時ファイルの利用

処理の途中で一時的なファイルを作成する必要が発生することがある。たとえば、すでに存在するファイルの中味を直接書き換える場合や、プログラムで生成したデータを別のプログラムに処理させたいのに別プログラムがデータをファイルでしか受け取らなかったりする場合は、一時ファイルが必要となる。ここでは、一時ファイルの作成と利用、適切な後始末の重要性について説明する。

6.4.1 一時ファイルとセキュリティ

処理の都合上、一時的なファイルを作成する場合、そのファイルをどこに置くかが問題となる。Unix 系のシステムでは慣習的に /tmp ディレクトリが一時ファイル作成場所として用意されている。ファイル名の衝突さえなければ自由な名前でファイルを作成することができるが、そのときにファイルの中味の漏洩に注意しなければならない。

たとえば、必ず /tmp/secret.tmp という一時ファイルを作成するプログラムがあったとしよう。プログラム起動者以外でも、そのファイル名でアクセスすれば読める危険がある。また仮に所有者以外が読み書きできないようファイル属性を変更していたとしても、同名のファイルを他人が作っておくことで、プログラムの実行の邪魔をしたりできる。

また、他人から次のようないたずらを仕掛けられる可能性もある。

```
% ln -s ~user/.zshrc /tmp/secret.tmp
```

/tmp/secret.tmp ファイルがない状態であれば、誰でも /tmp ディレクトリにこのシンボリックリンクを作れる。この状態で本人(user)がプログラムを起動すると、大事なファイル(この例では ~/.zshrc) を上書きして失うことに繋がる。

一時ファイルを作成・利用する場合には、以下の点に留意する必要がある。

1. **作らずに済むなら作らない。**
2. 予測可能なファイル名にしない。
3. 他者がアクセスできないファイル属性とする。
4. 不要になったら削除する。

1. は特に重要で、もし一時ファイルを作る目的が別プログラムにデータを渡すことなら、そのプログラムにデータを標準入力から読む機能があればそれを利用するように試みることが好ましい。

それでもなお一時ファイルを作成する場合は残りの 3 点に気を付けて行なう。これらを確実に行なうには、一時ファイルを作成するメソッドを正しい手順で利用する。

6.4.2　一時ファイル

RubyではTempfile[注12]を利用することで、一時ファイルの処理を確実に進めることができる。

```
require 'tempfile'
Tempfile.open(ファイルのベース名) {|fp|
  〜〜処理〜〜
}
```

以上のような流れにより、ファイル属性が0600（`-rw-------`）のファイルが、`"w+"`モードで開かれる。ブロックから抜けてcloseされると自動的に削除される。仮に、エラーでスクリプトが異常終了した場合も一時ファイルは削除される。

一時ファイルのファイル名が必要な場合は`fp.path`で得られる。また一時ファイルをオープンした状態で`fp.delete`とすると、即座にファイルを消して他のプロセスがアクセスする可能性を消すことができる。6.1.4節「ファイルの消えるタイミング」で述べたように、オープンしたままであれば元のプログラムからは引続きアクセスは可能である。

6.4.3　一時ディレクトリ

複数の一時ファイルを作成したり、ファイル名（の一部）を決まったものにしたい場合は、一時ファイルを格納する一時ディレクトリを作ればよい。

このときの手順を次に示す。

1. 競合する可能性の低い名前の一時ディレクトリを他者にアクセスできないモードで作る。
2. 作成したディレクトリにファイルを作り処理を済ます。
3. 一時ディレクトリ以下のファイルを削除する。

1.に関して、他のプログラム、あるいは同時に起動された同一プログラムが使う一時ファイルやディレクトリと名前が競合しないようにするための工夫として、一般的にはプログラム自身の名前とプロセスID（pid）を組み合わせた名前を利用することが多い。たとえば以下のような流れで決める。

注12　http://doc.ruby-lang.org/ja/2.1.0/class/Tempfile.html

```
myname = File.basename($0, ".rb")   # $0 はスクリプトの起動時のパス名
pid    = $$                         # $$ はプロセスIDを持つ組み込み変数
tmpdir = "/tmp/#{myname}-#{pid}"
```

そうして決めたディレクトリを作成する処理は、複数の処理に分けてはならない。たとえば以下のまずい例を考える。

```
pid = $$                            # PIDは $$ で得られる
tmpdir = "/tmp/foo-#{pid}.d"        # 一時ディレクトリ名を決める
if test(?d, tmpdir)                 # (1) 既存か確認
  abort "一時dir作成失敗"             # 既に存在したら異常終了
end
# (2)
Dir.mkdir(tmpdir)                   # なければ一時ディレクトリを作成し
# (3)
File.chmod(0600, tmpdir)            # すぐにchmodする
```

この場合、(1) のときにこれから作りたいディレクトリがないことを確認できたとしても、(2) のタイミングで別プロセスが同じ名前のディレクトリを作ってしまう可能性はある。また仮に、(2) の次の行まで無事進めてディレクトリ作成が成功しても (3) の一瞬を狙って別ユーザによる別プロセスがこのディレクトリをオープン (Dir.open) していれば、その後 chmod で別ユーザの読み取り属性を落としたとしても、そのプロセスは引続きディレクトリの中味を読み続けられる。

このような危険の可能性を排除するため、一時ディレクトリ作成はモード設定と合わせて単一アクションで行なう。この点に配慮して作成した例を示す。

リスト6.13●tmpdir.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-

require 'tmpdir'                  # Dir.tmpdir に必要
tmp = Dir.tmpdir                  # テンポラリディレクトリを得る 既定値=/tmp
me = File.basename($0, ".rb")     # スクリプト自身のベース名
dname = nil                       # 一時ディレクトリ名保存用変数
0.upto(9) do |n|                  # 10個の違うディレクトリ名で試行
  # ベース名とPIDに0〜9の番号を付けた名前とする
```

```
    dname = sprintf("%s/%s-%d.%d", tmp, me, $$, n)
    begin
      Dir.mkdir(dname, 0700)      # 他ユーザにアクセスできない属性で
      break
    rescue                        # mkdirが失敗したら
      dname = nil                 # ディレクトリ名を消しておく
    end
  end
  if dname && test(?d, dname)     # mkdirに成功した場合
    begin
      # ここに一時ディレクトリを利用した処理を書く
      Dir.chdir(dname) do
        printf("%s で一時ファイル作成\n", dname)
        open("hoge", "w") do |a| a.puts("hoge") end
        print `ls -laF`
        printf("1秒sleepします。C-cで止めても%sは消えます。\n", dname)
        sleep 1
      end
    ensure
      # ここには後始末(一時ファイル消去)処理を書く
      # system "/bin/rm -rf '#{dname}'" # 外部コマンドrmに任せる場合
      require 'fileutils'
      FileUtils.remove_entry_secure(dname, :force)
    end
  else
    abort "#{tmp} 内に一時ディレクトリを作れませんでした.
  環境変数 TMPDIR に書き込み可能なディレクトリを設定して再度起動してください。"
  end
```

このプログラムでは begin ～ ensure ～ end 構文を使っている。最初のブロックで異常事態が起きようと必ず ensure から end までのブロックの処理を行なう。実際に実行してみて一時ファイルが残らないことを確認してほしい。

なお、Ruby 1.8.7 以降では tmpdir ライブラリによる Dir.mktmpdir メソッドを使うことで、上記のような一時ディレクトリ作成・抹消処理ができる。上記 tmpdir.rb と同等の処理を Dir.mktmpdir を用いて書き換えたものを示す。

リスト6.14●mktmpdir.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-

require 'tmpdir'                # Dir.mktmpdir に必要
begin
  Dir.mktmpdir do |dname|
    Dir.chdir(dname) do
      printf("%s で一時ファイル作成\n", dname)
      open("hoge", "w") do |a| a.puts("hoge") end
      print `ls -laF`
      printf("1秒sleepします。C-cで止めても%sは消えます。\n", dname)
      sleep 1
    end
  end
rescue
  abort "#{tmp} 内に一時ディレクトリを作れませんでした.
環境変数 TMPDIR に書き込み可能なディレクトリを設定して再度起動してください。"
end
```

1 2 3 4 5 6 7 8 9 10

練習問題

6.1 リスト 6.8 について、下記 2 点の改良を施してみよ。

- 結果書き込み前にバックアップファイルに移動しておく。
- 書き込みできるか事前に確かめる。

6.2 複数のユーザのハイスコアをファイルに登録できるゲームプログラム `hiscore.rb` を作成せよ。以下のような仕様とする。

- 表示される 0 から 9 の乱数（`rand(10)` で得られる値）と同じ値をできるだけ早く入力し、その所要時間を競うゲームとする。
- 所要時間は値入力前と、正解入力後の `Time.now.to_f` の値の差で計算する。
- プログラムのインストール場所は、ある特定のトップディレクトリ（*PREFIX* で表す）内の `bin/` ディレクトリ、つまり *PREFIX*`/bin/` ディレクトリ、またハイスコアを記録するファイルは *PREFIX*`/share/` ディレクトリとする。
- ハイスコアファイルのファイル名は、プログラムのファイル名の `.rb` を `.score` に置き換えたものとする。
- ハイスコアファイルには、(1) ユーザ名、(2) 記録（時間）、(3) 達成日時、の 3 つを一組で記録し、上位 10 位まで記録する。

なお、このプログラムの実行に際しては *PREFIX*`/share` ディレクトリにファイルが書けることと、そこにあるスコアファイルには誰でも書き込めることを前提としてよい。

6.3 第 5 講の練習問題 3 ではドメインの存在を外部コマンドを使って確認しているが、これを Ruby 付属の resolv ライブラリを利用して行なうよう改めたプログラム `email_list2.rb` を作成せよ。
resolv ライブラリによる DNS レコード問い合わせは Resolv::DNS クラス[注 13] の

注 13　http://fiberstorm.gentei.org/~yuuji/tmp/class/2014/pf3/exercise/http://docs.ruby-lang.org/ja/2.1.0/class/Resolv=3a=3aDNS.html

getresource メソッドで以下のように行なう。

```
r = Resolv::DNS.new         # DNSリゾルバを生成する
r::getresource("example.com", Resolv::DNS::Resource::IN::ANY)
```

ただし、DNS の問い合わせに失敗すると例外を発生するため、getresource メソッドの呼び出しは、例外を捕捉できる begin 〜 end 〜 rescue 構文内で行なう必要がある。

第7講

周辺ツールの利用

7.1 フィルタコマンド

実用的なプログラムの作成においては、すべてをゼロから自分で作るのではなく、典型的な処理を既存のプログラムに任せることで圧倒的に早く目的を達成できる場合がある。

以降に示すコマンドは、いずれも標準入力（または指定したファイル）からテキストデータを読み込み、決められた処理を行なった結果を標準出力に書き出すフィルタコマンドである。これらをうまくパイプで繋いで利用することで、複雑な処理を一瞬で完了させることができる。

以下の説明では、省略できるものを大括弧 [ ] で括って表記する。また、記法説明で [ファイル群] とあるものは、ファイルを指定するとそのファイルを処理対象とし、ファイル指定を省略すると標準入力を処理対象とする。

7.1.1 nkf ー漢字コードの変換

日本語を含むテキストの文字コードを変換する。変換する文字コードはオプションで指定する。主なオプションを表 7.1 に示す。

表7.1●nkfのオプション

オプション	説明
-j	JIS コードに変換（デフォルト）
-e	日本語 EUC コードに変換
-s	Shift-JIS コードに変換
-w8	UTF-8 コードに変換
-w	UTF-8 コード（BOM 無）に変換
--in-place	変換対象ファイルを直接上書きして変換
-g	ファイルの文字コードの自動判別結果を出力する

いくつかの実行例を次に示す。

```
# ファイル foo をJISコードに変えたものを bar に保存
% nkf -j foo > bar
```

```
# ファイル hoge をEUCコードに変換したものを happy プログラムに渡す
% nkf -e hoge | ./happy

# ファイル bar をUTF-8コードに変換しそのまま上書きする
% nkf -w --in-place bar
```


7.1.2　egrep －検索パターンにマッチする行を抽出

指定したパターンにマッチする語を含む行を抽出して出力する。

```
egrep [オプション] パターン [ファイル群]
```

パターンには正規表現を利用する。*ファイル群*を省略すると標準入力を読んでパターン検索する。

よく使うオプションを表 7.2 に示す。

表7.2●egrepのオプション

オプション	説明
-i	アルファベットの大文字小文字を区別しない
-n	見つかった行の行番号を添えて出力する
-v	指定したパターンを含まない行のみ出力する
-l（小文字のエル）	パターンにマッチするファイルのファイル名のみ表示する
-w	指定したパターンを単語単位でマッチングさせる（例：egrep 'month' files... とした場合 month も monthly もマッチするが、egrep -w 'month' files... の場合は、monthly はマッチしない）

実行例を以下に示す。

```
# カレントディレクトリにあるファイルすべてから hoge というパターンを
# 含む行を探して表示
% egrep 'hoge' *

# カレントディレクトリにあるすべてのC言語ソースから printf を探す
% egrep 'printf' *.c
```

```
# カレントディレクトリにあるすべてのC言語ソースから printf を探す
# ただし fprintf にはマッチしないように単語単位で探す
% egrep -w 'printf' *.c

# カレントディレクトリより下にあるすべてのディレクトリにある
# C言語ソースから fgets というパターンを含むものを検索し、
# そのファイル名のみ表示
% egrep -l 'fgets' **/*.c
(**/*.c はカレントディレクトリ以下すべてにある *.c ファイル、
という意味でzshのみで利用できる)
```

7.1.3 wc ー行数・単語数・文字数の表示

入力の行数・単語数・文字数を数えて表示する。書式とオプションを次に示す。

```
wc [オプション] [ファイル群]
```

表7.3●wcのオプション

オプション	説明
-l	行数を表示（lines）
-w	単語数を表示（words）
-c	文字を表示（characters）

複数のオプションを組み合わせることもできる。たとえば wc -lc とすると、行数と文字数を表示することができる。

実行例を以下に示す。

```
# ~yuuji/zip_jp.txt から「山形県」を含むパターンを検索し、
# マッチした件数を数える
% nkf -e ~yuuji/zip_jp.txt | egrep '山形県' | wc -l
```

7.1.4　sed ーストリームエディタ

入力を加工したものを出力する。多くの処理ができるが、置換機能の使い方を覚えておけば十分だろう。

```
sed ［オプション］ 命令 ［ファイル群］
```

よく使うオプションを表 7.4 に示す。

表7.4●sedのオプション

オプション	説明
-e	次の引数を「命令」だと見なす（複数の命令を指定するときに有用）
-n	処理対象行をデフォルトで表示しない

「命令」の部分は多くのものが使えるが s 命令と p 命令だけ覚えておこう。s 命令は文字列の置換を行なう。

```
sed 's/置換前パターン/置換後文字列/'
```

とすると、入力行に「置換前パターン」に一致する部分があればそれを「置換後文字列」に置き換える。ただし、置き換えは 1 行につき 1 回のみしか行なわれない。1 行に現れる「置換パターン」をすべて置き換えてほしいときは、次のように g フラグを追加する（global の g）。

```
sed 's/置換前パターン/置換後文字列/g'
```

実行例を以下に示す。

```
# score.csv に現れる 太郎 を たろう に置き換えたものを出力
% cat score.csv | sed 's/太郎/たろう/'

# CSV形式のファイルをタブ区切りに変換する
% cat score.csv | sed 's/,/	/g'
（広い空白の部分はTAB文字で、コマンドラインではC-v TABで
 入力する）
```

p命令は、-nオプションと組み合わせて、特定の行だけ出力したい場合に利用する。次のようにすると、入力データが「行位置指定」にマッチする行のみ出力する。

```
sed -n '行位置指定p'
```

行位置指定には、/*正規表現*/または*行番号*が書ける。行位置指定をカンマで区切って列挙すると範囲指定になる。

```
# hoge.txt の中で Hello または hello にマッチする行のみ出力
% cat hoge.txt | sed -n '/[Hh]ello/p'
# でもこれなら grep '[Hh]ello' hoge.txt とやればいいので普通使わない
# ↓sedのp命令は行番号で使うと便利

# ~yuuji/zip_jp.txt のうち、50行目のみを表示
% cat ~yuuji/zip_jp.txt | sed -n 50p

# 本当に50行目というのがあってるか、cat -n で確認
% cat -n ~yuuji/zip_jp.txt | sed -n 50p

# ~yuuji/zip_jp.txt のうち、10～14行目のみを表示
% cat ~yuuji/zip_jp.txt | sed -n 10,14p

# ~/Mail/inbox/1 のうち、先頭から空行までを表示する
% cat ~/Mail/inbox/1 | sed -n '1,/^$/p'
(正規表現の先頭に来る ^ は行頭を意味し、
 正規表現の末尾に来る$は行末を意味する。
 よって /^$/という正規表現は、行頭のすぐ後ろに行末、つまり
 空行にマッチする)
```

7.1.5 awk ―パターンマッチと処理を行なう専用言語

awkは、入力を「空白で区切られたフィールドの集まり」と見なしてフィールドごとに分解した処理を簡単に書くことができる。

```
awk [オプション] awkプログラム文 [ファイル群]
```

*awk プログラム文*は ' ' で括る。ここには awk 言語の任意のプログラムを書けるが、print と printf だけ覚えておけばよい。*awk プログラム文*の部分では、入力行の各フィールドが $1、$2、$3、…、$NF で取り出せる。以下の例を見よ。

リスト7.1●sakata.txt（処理の元となるファイル）

```
998    9980032 サカタシ　アイオイチョウ
998    9980828 サカタシ　アキホチョウ
998    9980862 サカタシ　アケボノチョウ
998    9980021 サカタシ　アサヒシンマチ
998    9980875 サカタシ　アズマチョウ
998    9980055 サカタシ　イイモリヤマ
998    9980018 サカタシ　イズミチョウ
99977  9997774 サカタシ　イタド
998    9980046 サカタシ　イチバンチョウ
998    9980836 サカタシ　イリフネチョウ
```

awk にこのデータを与えると、第 1 フィールドが $1、第 2 フィールドが $2 のように自動的に変数に代入される。この入力から、第 2 フィールドと第 4 フィールドだけを表示したいときは以下のようにする。

```
% cat sakata.txt | awk '{print $2, $4}'
9980032 アイオイチョウ
9980828 アキホチョウ
9980862 アケボノチョウ
9980021 アサヒシンマチ
9980875 アズマチョウ
9980055 イイモリヤマ
9980018 イズミチョウ
9997774 イタド
9980046 イチバンチョウ
9980836 イリフネチョウ
```

-F オプションを使うとフィールドの区切り文字を変えることができる。

```
cat score.csv | awk -F, '{print $2, $4}'
```

とすると、区切り文字をカンマ（,）としてフィールド分割を行なう。単純なCSVファイルを処理するときはawk -F,によって特定のフィールドのみを対象とした処理が簡単に書ける。

7.1.6 sort ー入力レコードのソート

1行1レコードとして入力行をソートした結果を出力する。標準では行頭にあるフィールドを文字コード順にソートする。sortでよく使うオプションは以下のとおり。

表7.5●sortのオプション

オプション	説明
-k *N*	ソートする基準（キー）となるフィールドを第*N*フィールドにする（*N*は1から始まる整数）。デフォルトは第1フィールドで、フィールド番号は1から数え始める。
-t *C*	フィールドの区切り文字を*C*にする。CSV形式であれば-t ,と指定する。
-n	該当フィールドを数値だと見なしてソートする。
-r	逆順（降順）にソートする。

リスト7.2●元となるデータ（score.txt）

```
山田太郎	50	70	20
公益太郎	90	80	70
飯森花子	91	79	72
鶴岡一人	60	60	40
酒田三吉	52	70	80
三川一二三	12	34	99
```

この構成は以下のようになっている。

第1フィールド　　氏名（文字列）
第2フィールド　　数学得点（数値）
第3フィールド　　英語得点（数値）
第4フィールド　　国語得点（数値）

sortコマンドによる並べ替えは以下のようになる。

```
# 漢字氏名でソート
% cat score.txt | sort

# 数学得点でソート
% cat score.txt | sort -n -k 2

# 英語得点でソート
% cat score.txt | sort -n -k 3

# 国語得点の高い順にソート
% cat score.txt | sort -nr -k 4
```

元となるデータがCSV形式だった場合はsortの -t オプションで区切り文字をカンマに変えればよい。

リスト7.3●score.csv

```
山田太郎,50,70,20
公益太郎,90,80,70
飯森花子,91,79,72
鶴岡一人,60,60,40
酒田三吉,52,70,80
三川一二三,12,34,99
```

```
# 数学得点でソート
% cat score.csv | sort -n -t , -k 2

# 国語得点の高い順にソート
% cat score.csv | sort -nr -t , -k 4

# 国語得点の高い順にソートしたものをTAB区切りに変換して出力
% cat score.csv | sort -nr -t , -k 4 | sed 's/,/ /g'
(sed命令にある広い空白の部分はC-v TABとタイプして入力する)

# 数学得点の高い順にソートしたものを、氏名幅20桁にきれいに揃えて出力
% cat score.csv | sort -nr -t , -k 4 | \
      awk -F, '{printf "%-20s\t%d %d %d\n", $1, $2, $3, $4}'
```

7.1.7 uniq ー重複行の削除と重複数数え上げ

標準入力(または指定したファイル)で連続して同じ行があった場合にそれを1行のみにする。有用な -c と -u オプションは覚えておくとよい。

表7.6●uniqのオプション

オプション	説明
-c	1行にまとめるときに、行頭に何行をまとめたかを付加する。
-u	連続する行でなくてもすでに登場した行と同じものは出力しない。

たとえば、入力が

```
abc
abc
def
def
def
hoge
def
```

のときの uniq -c の結果は次のようになる。

```
   2 abc
   3 def
   1 hoge
   1 def
```

これは頻度表を作るときに有用である。登場数を調べたい1つの項目が1行になるようにしたファイルから登場数ランキングを求めるときには、たとえば次のようにする。

```
% cat one-item-per-line.txt | sort | uniq -c | sort -nr
```

7.1.8 tr ー文字種変換

標準入力の中に登場する各文字を指定した文字にすべて変換する。大文字を小文字にしたり、改行文字と空白を相互に変換するときに用いる。

大文字から小文字へ変換する例：

```
% cat srcfile | tr '[A-Z]' '[a-z]' > destfile
```

空白を改行に全置換する例：

```
% cat srcfile | tr ' ' '\n' > destfile
```

-d オプションに続けて文字（または文字範囲）を指定すると、入力側からその文字を削除する。次の例は MS-DOS 改行（CR+LF）を Unix 改行（LF）に変換する。

```
% cat dosfile | tr -d '\r' > unixfile
```

7.1.9 head ー先頭表示

標準入力（または指定したファイル）の先頭 10 行を出力する。-*数字*オプションで出力する行数を変えることができる。

```
# foo.txtの先頭10行を表示
% head foo.txt

# カレントディレクトリにある *.c ファイルすべての
# 先頭を表示してlessで読む
% head *.c | less
# less は、SPCで進んでbで戻る。qで終了

# score.csvを数学得点でソートした結果の先頭3行を表示
% cat score.csv | sort -n -t, -k2 | head -3
```

7.1.10 tail ー末尾表示

head とは反対に、標準入力（または指定したファイル）の末尾 10 行を出力する。

tail コマンドは増えゆくファイルの末尾をつねに監視し続けるときに利用する。これには -f オプションを指定する。

```
% tail -f filename
```

Web サーバやメイルサーバプログラムのログファイルを監視するときに有用である。

7.2 複雑な処理を行なう例

複雑な処理を実行するためにフィルタコマンドを組み合わせていく様子を、7.1.6 節で使った成績ファイルの内容を合計点の高い順に並べ替える作業を例に説明しよう。

リスト7.4●score.csv（再掲）

```
山田太郎,50,70,20
公益太郎,90,80,70
飯森花子,91,79,72
鶴岡一人,60,60,40
酒田三吉,52,70,80
三川一二三,12,34,99
```

これは CSV ファイルなので、各フィールドに分解するには awk コマンドを使えばよい。区切りがカンマなので、awk のオプションに -F, を指定すればフィールド分解が適切に行なわれる。「cat score.csv | awk -F,…」としたときに awk 変数に入る各フィールドは次のようになる。

表7.7●awk変数に入る各フィールド

$1 に入るもの	$2 に入るもの	$3 に入るもの	$4 に入るもの
山田太郎	50	70	20
公益太郎	90	80	70
飯森花子	91	79	72
鶴岡一人	60	60	40
酒田三吉	52	70	80
三川一二三	12	36	99

各レコードについて合計点を求めるには、$2、$3、$4 を足せばよい。awk 言語は文字列と数値の区別があいまいである。本来 $2、$3、$4 は入力レコードの部分文字列なので、どれも文字列なのだが、数字だけの文字列なら足し算をすると数値同士と見なして足し算してくれる。つまり、$2+$3+$4 と書くだけで 3 つの数の合計が得られる。この性質を利用すると、素点のみが記された CSV に合計点を追加する awk コマンドラインは以下のようになる。

```
% cat score.csv \
| awk -F, '{printf "%s,%d,%d,%d,%d\n", $1, $2, $3, $4, $2+$3+$4}'
山田太郎,50,70,20,140
公益太郎,90,80,70,240
飯森花子,91,79,72,242
鶴岡一人,60,60,40,160
酒田三吉,52,70,80,202
三川一二三,12,34,99,145
```

printf は Ruby のものとほとんど同じ書式変換を行なうもので、上記実行結果が得られる。これで CSV の第 5 フィールドに合計点が追加された。あとは、sort コマンドで第 5 フィールドを数値と見なして並べ替えればよい。sort コマンドもデフォルトではフィールド区切りが空白文字なので、「-t ,」としてカンマでフィールド分割させる。数値と見なして並べ替えるには -n オプションを指定する。

```
% cat score.csv \
| awk -F, '{printf "%s,%d,%d,%d,%d\n", $1, $2, $3, $4, $2+$3+$4}' \
| sort -t , -n -k 5
山田太郎,50,70,20,140
三川一二三,12,34,99,145
```

```
鶴岡一人,60,60,40,160
酒田三吉,52,70,80,202
公益太郎,90,80,70,240
飯森花子,91,79,72,242
```

ここで得られた結果は小さい順（昇順）に並んでいる。合計点の高い順に並べるのが目的なので、大きい順（降順）に並べるために -r オプションを付けて並べ替えを逆にする。

```
% cat score.csv \
| awk -F, '{printf "%s,%d,%d,%d,%d\n", $1, $2, $3, $4, $2+$3+$4}' \
| sort -r -t , -n -k 5
飯森花子,91,79,72,242
公益太郎,90,80,70,240
酒田三吉,52,70,80,202
鶴岡一人,60,60,40,160
三川一二三,12,34,99,145
山田太郎,50,70,20,140
```

さらにこの結果を cat -n に渡せば、次のように順位を付けることができる。

```
% cat score.csv \
| awk -F, '{printf "%s,%d,%d,%d,%d\n", $1, $2, $3, $4, $2+$3+$4}' \
| sort -r -t , -n -k 5 \
| cat -n
     1	飯森花子,91,79,72,242
     2	公益太郎,90,80,70,240
     3	酒田三吉,52,70,80,202
     4	鶴岡一人,60,60,40,160
     5	三川一二三,12,34,99,145
     6	山田太郎,50,70,20,140
```

先頭の空白が多すぎだと思ったら、sed コマンドで文字列の置換（s 命令）を使ってスペースを 4 つ削る。

```
% cat score.csv \
| awk -F, '{printf "%s,%d,%d,%d,%d\n", $1, $2, $3, $4, $2+$3+$4}' \
| sort -r -t , -n -k 5 \
| cat -n \
```

```
| sed 's/    //'     (← s/ の後ろのスペースは4つ)
 1       飯森花子,91,79,72,242
 2       公益太郎,90,80,70,240
 3       酒田三吉,52,70,80,202
 4       鶴岡一人,60,60,40,160
 5       三川一二三,12,34,99,145
 6       山田太郎,50,70,20,140
```

ついでに、カンマも TAB 文字に変えてしまう。再び sed の置換（s 命令）を使って TAB 文字に変換する。

```
% cat score.csv \
| awk -F, '{printf "%s,%d,%d,%d,%d\n", $1, $2, $3, $4, $2+$3+$4}' \
| sort -r -t , -n -k 5 \
| cat -n \
| sed 's/    //' \
| sed 's/,/    /g'    (← の部分はC-v TABで入力)
 1       飯森花子        91      79      72      242
 2       公益太郎        90      80      70      240
 3       酒田三吉        52      70      80      202
 4       鶴岡一人        60      60      40      160
 5       三川一二三      12      34      99      145
 6       山田太郎        50      70      20      140
```

このように、プログラムを書くことなく並べ替えと整形を行なうことができた。

練習問題

7.1 /etc/group ファイルには、システムのユーザが属するグループの情報が記されている。

例：

```
wheel:*:0:root,yuuji
daemon:*:1:daemon
kmem:*:2:root
sys:*:3:root
tty:*:4:root
operator:*:5:root,yuuji
```

コロン区切りのこの形式のファイルから、第 1 カラム（グループ名）と第 4 カラム（所属ユーザ）を抽出して以下のように出力したい。

```
% ./grp.rb
wheel = root,yuuji
daemon = daemon
kmem = root
sys = root
tty = root
operator = root,yuuji
```

1. Ruby プログラムでこれを行なう grp.rb を作成せよ。
2. awk を用いてこれを行なうコマンドラインを作成せよ。
3. sed を用いてこれを行なうコマンドラインを作成せよ。

なお、以上いずれの場合も /etc/group ファイルのすべての行が正しくコロン区切りで正しい数のカラム数のレコードを含んでいるものとしてよい。

7.2 /etc/passwd ファイルには、システムのアカウント情報がコロン区切りで登録されている。

例：

```
root:*:0:0:Charlie &:/root:/bin/zsh
toor:*:0:0:Bourne-again Superuser:/root:/bin/sh
daemon:*:1:1:The devil himself:/:/sbin/nologin
yuuji:*:20200:100:HIROSE Yuuji:/home/yuuji:/bin/zsh
```

第 3 カラムはユーザ ID（UID）、第 4 カラムはグループ ID（GID）で、0 から始まる整数値を取る。第 6 カラムはユーザのホームディレクトリ（HomeDir）である。ユーザ ID が 10000 以上 60000 未満のすべてのユーザの UID、GID、HomeDir に対して、

```
cp -r /etc/skel/. HomeDir && chown -R UID:GID HomeDir
```

のような行を出力するコマンドラインを作成せよ。なお、ユーザ ID が 10000 以上 60000 未満のユーザ名だけを取り出すには以下のようにする。

```
% cat /etc/passwd | awk -F: '$3>=10000 && $3<60000{print $1}'
```

7.3 Apache で稼働している Web サーバに通常の Web ページアクセスがあると、アクセスログには以下のような 1 行が記録される（紙面の都合で折り返し表示しているが実際には 1 行である）。

```
fiberstorm.gentei.org - - [14/Jun/2014:11:23:43 +0900] "GET /~yuuji/diary/index.
html HTTP/1.1" 200 1760 "http://www.gentei.org/~yuuji/index.html"
```

「GET」の直後にある表記が取得された URL のパス名相当部分である。このような形式のログを解析し、アクセスされた回数の多い順にパス名を出力するコマンドラインを作成せよ。アクセスログのファイル名は access_log とする。

第8講

シェルの活用

これまではプログラムを起動するための場としてしか利用していなかったシェルだが、シェル自身も変数や制御構造を持つインタプリタであり、きわめて高度な作業効率を誇る。たとえば、「"Hello" と書かれたファイル hello.txt を作成する」というとき、Ruby であれば次のように書ける。

```
#!/usr/local/bin/ruby
open("hello.txt", "w"){|h| h.puts "Hello"}
```

これは数あるプログラミング言語のうちで相当短く書ける部類なのだが、シェルであればさらに少ない記述量で書ける。

```
#!/bin/sh
echo Hello > hello.txt
```

「ファイルを開いてデータを書き込む」という操作がたった 1 字の記号「>」で完了する。その他にも、1 字か 2 字の記号によく使う条件分岐の意味が込められているなど、活用すれば圧倒的に短くスクリプトが作成できる有用な構文がシェルには存在する。

ここでは、sh 系（sh、ksh、bash、zsh）でプログラミングを行なう上で知っておくべきことの要点をまとめる。また本書の見本環境として利用している zsh にはプログラムの記述効率を高める記法が多数あるので、一部は zsh のみと明記した上でそれを記した。

zsh だけでなく、sh 系シェルの言語としての仕様はバージョン間の差異がほとんどないため、長期に渡って利用するものを書きやすい。Ruby や Python のような先鋭言語は活発に仕様拡張が行なわれるため、将来の仕様変更で自作スクリプトの書き直しを迫られる可能性がどうしてもつきまとう。長期間の利用が見込まれる「息の長い」スクリプトを作る必要がある場合は sh 系の言語の利用も選択肢の一つとして加えた方がよいだろう。

8.1 シェル変数

8.1.1　定義と参照

変数定義は次の形式で行なう。

変数=値

イコールの前後に空白は許されない。シェル変数は慣習的に小文字を用いる。値は定義したシェルのみが持ち、他のシェルやプロセスからは見えない。変数一覧はシェルの内部コマンド set にて得られる。

```
% set
HOME=/home/tar
HOST=hornet
(以下略)
```

定義したシェル変数の値の参照は次のいずれかで行なう。

$*変数*
${*変数*}

単語区切りが明確なときのみ、{　} は不要である。たとえば、シェル変数 foo の値の直後に文字列 s を付けて出力したい場合は次のようにする。

```
% foo=cat                (代入)
% echo $foo              (変数展開)
cat
% echo $foos             (変数 foos を展開)

% echo ${foo}s           (変数 foo を展開)
cats
```

この例で「echo $foos」とした部分は、変数 foos の展開を試みているが、この時点で変数 foos は未定義である。sh では未定義変数を参照してもエラーにはならず、単に空文字列を返す。ただし、「空文字列（""）を値に持つ変数」と「未定義変数」を区別できないわけではない。両者を区別して扱いたい場合は、次項で説明する条件付き展開を用いる。

8.1.2 条件付き展開

ある変数に値が定義されているかどうかによって展開結果が変わる記法がある。おおむね ${ *変数 記号 値* } のような形で記すもので、代表的なものを次に示す。

表8.1●代表的な条件付き展開

条件付き展開	説明
${ *変数* :- *値* }	デフォルト値。*変数*の値が未定義か空文字列なら*値*に展開する。そうでなければ*変数*の値に展開する。
${ *変数* := *値* }	デフォルト値と代入。*変数*の値が未定義か空文字列なら*値*に展開しつつ*変数*にもその値を代入する。そうでなければ*変数*の値に展開する。
${ *変数* :?[*文*]}	値保持強制。*変数*の値が未定義か空文字列なら*文*（省略時はデフォルトのエラーメッセージ）を標準エラー出力に出し、スクリプトを終了する。そうでなければ変数の値に展開する。
${ *変数* :+ *値* }	値保持連動。*変数*の値が未定義か空文字列なら空文字列（""）に展開する。そうでなければ*値*に展開する。

なお、上記の記号はどれもコロン（:）で始まっているが、コロンを省略して書くこともでき、その場合は「未定義か空文字列なら」ではなく、「未定義なら」に働きが変わる。

8.1.3 特殊な展開

変数の値を文字列加工した結果に展開する規則が使える。ここではスクリプト処理で頻繁に用いるパス名の操作に利用できる記法を記す。以下の表記の*パターン*はシェルのファイル名展開で使えるパターンで、*、?、[ ] の記号が使える。

表8.2●特殊な展開

展開規則	説明
${ *変数* %*パターン* }	末尾最短マッチ除去。*変数*の値の文字列の末尾から*パターン*にマッチする最短部分を取り除く。
${ *変数* %%*パターン* }	末尾最長マッチ除去。*変数*の値の文字列の末尾から*パターン*にマッチする最長部分を取り除く。
${ *変数* #*パターン* }	先頭最短マッチ除去。*変数*の値の文字列の先頭から*パターン*にマッチする最短部分を取り除く。
${ *変数* ##*パターン* }	先頭最長マッチ除去。*変数*の値の文字列の先頭から*パターン*にマッチする最長部分を取り除く。

これを用いて、パス名からディレクトリ名を得たり、ディレクトリ部分を除去したベース名を得られる。次の例は、シェル変数 foo に /usr/local/bin/cmd.sh という値が代入されている場合の展開の様子を示したものである。

```
% echo ${foo%/*}
/usr/local/bin
% echo ${foo#*/}
usr/local/bin/cmd.sh
% echo ${foo##*/}
cmd.sh
```

メイルアドレスが代入されたシェル変数から、ローカル部（@ より前の部分）とドメイン部（@ より後ろの部分）を取り出す場合などに利用できる。

8.2 環境変数

8.2.1　シェル変数と環境変数

シェル変数は起動しているシェルプロセスのみで有効なものである。そのうち、指定した変数をそのシェルから起動されるすべての子孫プロセスに継承させることができる。別プロセス

に引き継がれる変数のことを**環境変数**といい、すべてのプロセスがこの情報を親プロセスから引き継いで始まる。ただし環境変数表はプロセスごとに独立しているため、子プロセスで設定した変数は親プロセスには反映されない。

sh ではシェル変数を export することで環境変数化でき、そのシェルから起動するすべての子孫プロセスにコピーされて渡される。慣習的に環境変数の名前は大文字が用いられる。環境変数の値を出力するための外部コマンド printenv を用いて挙動を示す。

```
% FOO=Hello              # シェル変数として定義
% printenv FOO           # 何も出力されない
% export FOO             # exportする
% printenv FOO           # 環境変数なのでOK
Hello
```

export は指定したシェル変数をそれ以後ずっと export し続ける。export を使わずに、次のように変数代入形式の後ろに続けてコマンド起動の文を書けば、そこで起動されるコマンドだけに環境変数を与えることができる。

```
% 変数=値 コマンド...
```

このとき行なわれる変数設定は、シェル自身の変数設定には影響を与えない。

```
% printenv BAR               # 何も設定していないので出ない
% BAR=yes printenv BAR       # printenvコマンドのときだけ設定される
yes
% echo $BAR                  # 元のシェルの変数には無影響

%
```

変数 = *値*の組は空白で区切って何個でも与えられる。この起動方法を利用すると、コマンドへの情報授受が簡単確実に行なえる。たとえば

> 一時ファイルを作るディレクトリとして、特に指定がなければ /tmp、環境変数 TMPDIR が設定されている場合はその値のディレクトリを利用する。

のように振る舞うプログラムが存在するとして、そのようなプログラムに臨時で特定のディレクトリを使わせたい場合は次のように起動すればよい。

```
% TMPDIR=/var/tmp command...
```

実際に上記のように環境変数で挙動を変えるプログラムを自分で書く場合は、次のような書き出しで作成できる。

```
#!/bin/sh
tmpdir=${TMPDIR:-/tmp}
```

Ruby の場合であれば次のようにする。

```
#!/usr/local/bin/ruby
tmpdir = ENV["TMPDIR"] || "/tmp"
```


8.3 コマンド置換

■ `cmdline`（バッククォート）

Ruby のバッククォート（5.1.2 節参照）と同様、内部のコマンド *cmdline* を起動した結果の文字列に置換される。

■ $(*cmdline*)

バッククォートと同様、コマンドを起動した結果の文字列に置換される。なお、バッククォートと違いネストすることができる。

自分が差出人のメイルから、送信先となっている頻度が一番多い人にメイルを送る手順を例に示す。

```
% cd ~/Mail/inbox
% grep -l "Return-Path:.*$USER" *
(マッチするファイル名一覧が出る)
% grep '^To:' $(grep -l "Return-Path.*$USER" *)
(自分が差出人のファイルから To: ヘッダの行を抽出)
% grep '^To:' $(grep -l "Return-Path.*$USER" *) | \
  ruby -pe 'sub(/.*[ <](.+@[^>]*)>?/,"\\1")'
(送信先のアドレス一覧が出る)
% grep '^To:' $(grep -l "Return-Path.*$USER" *) | \
  ruby -pe 'sub(/.*[ <](.+@[^>]*)>?/,"\\1")' | \
  sort | uniq -c | sort -nr | head -1 | awk '{print $2}'
(頻度1位の人のアドレスが出る)
% echo 1位です! | nkf -j | \
  Mail -s congratulations $(grep '^To:' $(grep -l "Return-Path.*$USER" *) | \
  ruby -pe 'sub(/.*[ <](.+@[^>]*)>?/,"\\1")' | \
  sort | uniq -c | sort -nr | head -1 | awk '{print $2}')
```

8.4 算術展開

一部の sh を除くほぼすべての sh 系のシェルでは、$((*数式*)) の形で整数の簡単な演算が行なえる。数式の部分には数値を値に持つシェル変数が使え、その場合 $ を付けても付けなくてもよい。

```
% i=1024                 (シェル変数iに代入)
% echo $((i+$i))         (iでも$iでもよい)
2048                     (1024+1024=2048)
% echo $((3*i*i/512))
6144
```

8.5 ブレース展開

中括弧 { } の内部にカンマで区切った文字列を列挙すると、それらを開いた文字列に展開する。前後に文字列を付けるとそれらと結合した上で展開する。

```
% echo {a,b,c}
a b c
% echo file-{a,b,c}
file-a file-b file-c
% echo hoge.rb{,bak}
hoge.rb hoge.rb.bak
% echo cp Ruby/kadai1/hogehoge.rb{,.bak}
cp Ruby/kadai1/hogehoge.rb Ruby/kadai1/hogehoge.rb.bak
```

zsh 拡張として数値範囲を指定した展開ができる。

```
% echo {1..20}
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
% echo file{1..20}
file1 file2 file3 file4 file5 file6 file7 file8 file9 file10 file11
file12 file13 file14 file15 file16 file17 file18 file19 file20
(空のファイルを作るためにtouchコマンドを用いる)
% touch file{1..999}
(lsで確認後消去)
% ls
rm file{1..999}
```

8.6 制御構造

あるコマンドの実行が成功するか（終了値 0）失敗するかによって分岐したり、繰り返し処理を行なったりできる。

■ *cmdline*$_1$ && *cmdline*$_2$

cmdline$_1$ の実行が成功したときのみ *cmdline*$_2$ を実行する。

```
% cd
% touch myprogram.rb && chmod +x myprogram.rb
(ホームディレクトリには作れるので chmod +x myprogram.rb される)
% rm myprogram.rb
% touch /myprogram.rb && chmod +x /myprogram.rb
(ルートディレクトリには作れないので chmod されない)
```

■ *cmdline*$_1$ || *cmdline*$_2$

cmdline$_1$ の実行が失敗したときのみ *cmdline*$_2$ を実行する。

■ if *cmdline*$_1$; then ～ elif *cmdline*$_2$; then ～ else ～ fi

cmdline$_1$ の実行が成功したときに then に続くブロックを評価する。いずれのコマンドも失敗したときは else ブロックを評価する。

```
#!/bin/sh
if touch hogehoge.rb
then
  echo ./hogehoge.rb 作れました
elif touch /tmp/hogehoge.rb
  echo /tmp/hogehoge.rb 作れました
else
  echo 失敗しました。やめ。
  exit
fi
```

コマンドの部分には条件判断を行なうだけのコマンドを使うことが多い。典型的には /bin/test コマンドでこれはファイルの存在を確かめたりできる。Ruby の test 関数[注1]は外部コマンド test に由来する。

```
#!/bin/sh
if /bin/test -f hogehoge.rb
then
  echo hogehoge.rb ファイル、あります。
else
  echo hogehoge.rb ファイル、ありません。
fi
```

条件式に見えやすいよう、test コマンドには [という名前のハードリンクが張られているので、これを用いて書き換えると以下のようになる。

```
#!/bin/sh
if [ -f hogehoge.rb ]; then
  echo hogehoge.rb ファイル、あります。
else
  echo hogehoge.rb ファイル、ありません。
fi
```

test の代わりに [で呼び出したときは行の終わりに] を付ける。

■ while *cmdline*; do ～; done

cmdline が成功を返す間ブロックを評価し続ける。*cmdline* の部分には test コマンドや、常に成功する /bin/true コマンドを用いることが多い。

```
#!/bin/sh
while true; do
  echo 誰か止めて～
  sleep 1
done
```

注 1　http://doc.ruby-lang.org/ja/2.1.0/class/Kernel.html#M_TEST

■ for *変数* in *語群* ; do ～ ; done

語群（を展開した結果）を先頭要素から順に*変数*に代入しつつブロックを繰り返す。

次の例はカレントディレクトリに含まれる *.rb ファイルすべてのバックアップファイルを作成する。

```
% for f in *.rb; do
cp $f $f.bak
done
```

8.7 シェル関数

一連の手続をまとめて関数化できる。シェル関数の定義は次の形式で行なう。

```
関数名() {
  …定義本体…
}
```

関数への引数は $1、$2、$3、…で受け取る。$* はすべての引数を表す。

```
# 定義する
% foo() {
 echo 123: $1 $2 $3
 echo all: $*
}
# 呼び出す
% foo a b c d e f g
123: a b c
all: a b c d e f g
```

8.8 グロッビング

ファイル名に対するパターンマッチングを**グロッビング**という。正規表現とは規則が違う。

■ ?

任意の 1 字にマッチ

■ *

0 字以上の任意の文字列にマッチ

■ [*文字クラス*]

*文字クラス*のいずれかの 1 字にマッチ。

zsh 拡張として次のパターンも使える。

■ **/

ディレクトリを再帰的に検索

■ <*整数* 1- *整数* 2>

数値的に*整数* 1 以上*整数* 2 以下の文字列にマッチする。上限のみ、下限のみでもよい。たとえば、カレントディレクトリに 15 個のファイル、`file1.rb`、`file2.rb`、……、`file15.rb` がある場合、

```
file<4-12>.rb
```

は、`file4.rb`、`file5.rb`、……、`file12.rb` の 9 個のファイル、

```
file<-9>.rb
```

は、file1.rb、file2.rb、……、file9.rb の 9 個のファイル、

```
file<8->.rb
```

は、file8.rb、file9.rb、……、file15.rb の 8 個のファイルにマッチする。
　もちろん存在しないファイル名は出てこない。

■ パターン$_1$~ パターン$_2$

　パターン$_1$にマッチするものからパターン$_2$にマッチするものを除外する。たとえば、上述のfile1.rb、file2.rb、……、file15.rb の 15 個のファイルがある場合、

```
file??.*~*15*
```

と指定すると、"file" の後ろに任意の 2 字が来て、そのあとピリオドと任意文字列が来るファイル名すべてをまず選び、そこから途中に "15" という文字列が現れるものを除外するので、file10.rb、file11.rb、file12.rb、file13.rb、file14.rb がマッチする。

8.9 入出力

　シェルは標準入出力を処理するフィルタとしても振る舞える。シェルが直接標準入力を読むには read、標準出力に送るには echo を用いる。

■ read *変数* ...

　標準入力から 1 行読み込み、変数に代入する。変数を 2 個以上指定した場合は、空白文字でフィールド分割した各フィールドを先頭の変数から順次代入する。フィールドの方が多い場合

は最後の変数に残りすべてが代入される。

```
% read x
foo bar baz
x -> "foo bar baz"
% read x y
foo bar baz
x -> "foo"
y -> "bar baz"
% read x y z
foo bar baz
x -> "foo"
y -> "bar"
z -> "baz"
```

フィールド区切りを空白以外に変更するには、シェル変数 IFS に区切り文字を列挙する。CSV ファイルを読み取って処理する例を示す。

リスト8.1●score.csv

```
山田太郎,50,70,20
公益太郎,90,80,70
飯森花子,91,79,72
鶴岡一人,60,60,40
酒田三吉,52,70,80
三川一二三,12,34,99
```

```
% cat score.csv | while IFS=, read name math eng jp
do
  sum=$((math + eng + jp))
  echo $name さんは合計 $sum 点です。
  if [ $sum -lt 200 ]; then
    echo "*** $name さんは赤点です ***"
  fi
done > result.txt
% cat result.txt
山田太郎 さんは合計 140 点です。
*** 山田太郎 さんは赤点です ***
```

1
2
3
4
5
6
7
8
9
10

```
公益太郎 さんは合計 240 点です。
飯森花子 さんは合計 242 点です。
鶴岡一人 さんは合計 160 点です。
*** 鶴岡一人 さんは赤点です ***
酒田三吉 さんは合計 202 点です。
三川一二三 さんは合計 145 点です。
*** 三川一二三 さんは赤点です ***
```

練習問題

8.1 練習問題 7.1 と同じ働きをするシェルスクリプト grp.sh を、外部コマンドを使わずに作成せよ。

8.2 練習問題 7.2 では、/etc/passwd の第 3（UID）、第 4（GUID）、第 6（HomeDir）カラムを抽出するコマンドラインを生成した。これにならい、ユーザ ID が 10000 以上 60000 未満のユーザのホームディレクトリ（HomeDir）が存在したら、

```
cp -r /etc/skel/. HomeDir && chown -R UID:GID HomeDir
```

を実際に行なうようなシェルスクリプト、cpskel.sh を作成せよ。なお、UID の大小比較には test コマンドを用いる。test の条件記述で、2 つの条件を「かつ」で結ぶには、-a を用いて以下のように記述する。

```
if [ 条件1 -a 条件2 ]; then
  # 該当する場合の処理
fi
```

また、ディレクトリが存在するかどうかは test -d *ディレクトリ*で検査できる。

8.3 ex エディタでファイルを開くと、以下のようなエディタコマンド入力待ちのプロンプト（:）が現れる。

```
% ex foo.txt
foo.txt: unmodified: line 8
:
```

このプロンプトで「%! *Command*」と入力すると、編集バッファの中味をすべて外部コマンド *Command* の標準入力に渡し、出力された内容でバッファを置き換える。このあと「wq」とコマンド入力すると、新しい内容をファイルに書き込んで終了する。したがっ

て、たとえば ex エディタでファイルを開いた後に

```
%! nkf -j Enter
wq Enter
```

と続けて入力すると、ファイルの中味を JIS コードに変換した結果で上書き保存することができる。ex エディタは編集コマンドを標準入力から読むため、以下のように実行すると foo.txt を JIS コードに上書き変換できる。

```
% echo "%! nkf -j
wq" | ex foo.txt
```

以上のことを利用して、第 1 引数に指定したコマンドラインをすべて第 2 引数以後のファイル（群）に適用し、変換結果を上書き保存する次のようなシェルスクリプト（overwrite.sh）を、ex を利用し作成せよ。

```
% ./overwrite.sh "フィルタコマンド" File(s)
```

たとえば、カレントディレクトリにあるすべての *.rb ファイルの「coding: euc-jp」を「coding: utf-8」に置換し、ファイルの文字コード自体を utf-8 に変換するには、次のようにすればできるものとする。

```
% ./overwrite.sh "sed 's/coding: euc-jp/coding: utf-8/'|nkf -w" *.rb
```

第9講

自由度の高いCUI

9.1 cursesライブラリ

これまで作成したプログラムを利用する場合、利用者がプログラムに対して行なうアクションは「文字列を入力してEnterを押す」という一方向のものばかりであった。また、画面への出力も上から下へと流れるものしかなかったが、**cursesライブラリ**[注1]を用いると、より高度で自由度の高い対話的プログラムが作れる。

なお、Rubyバージョン2.1からは、cursesが標準ライブラリから外れたため、利用するためには追加インストールする必要がある。まず、利用中のRubyでcursesが使えるか確認するため、以下のように起動する。

```
% ruby -rcurses -e ''
```

もし、これでエラーメッセージが出るようであれば、システムのRubyにcursesライブラリが組み込まれていない。その場合はスーパーユーザ等十分な権限でインストールを行なう。一例を示す。

```
% sudo gem install curses
```

9.2 cursesを利用してできること

cursesをプログラム内に導入し、利用宣言すると以下のことが可能になる。

- 画面の好きな位置に文字列を出せる。
- Returnを押さなくてもすぐにキー入力できる。
- 制限時間付きのキー入力が可能になる。

注1　http://docs.ruby-lang.org/ja/2.0.0/library/curses.html

その一方、画面出力・キー入力がすべて curses のものになるため、普通に標準入出力を用いているプログラムで可能な以下のことが**できなくなる**。

- puts、print、printf などを使った文字列出力（期待どおりの場所に出ない）。
- 明示的に設定しなければ出力画面の自動スクロールがされない（最下行で止まる）。
- gets などの入力関数（データを送れない）。

ただし、いずれも curses 専用の制御メソッドを利用するよう気を付ければ問題ない。

9.3 curses の導入

curses を利用したプログラムは以下のような流れで作成する。

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'curses'
include Curses

init_screen                     # スクリーンを初期化する

begin
  ～ 必要な処理本体 ～
ensure
  close_screen                  # スクリーンを元に戻す
end
```

curses の機能を提供するメソッドは Curses::*メソッド名*によって呼び出す。ただし、上のように記述すれば 4 行目の include 文によって Curses:: の接頭辞なしでメソッド呼び出しが可能になる。

curses では端末画面のさまざまな設定値を変更するため、最後に設定値を元に戻す処理を行なわないと、その後の端末操作に異常を来たす場合がある。このため、プログラムが異常終了

した場合でも確実に close_screen が行なわれるよう、ensure ブロックで確実に処理する。

9.4 画面制御

curses を用いると、仮想端末の画面を自在にレイアウトするプログラムを容易に作ることができる。たとえば、画面上「5 行目、10 桁目」に「こんにちは」と表示するプログラムは次のようになる。

リスト9.1●cur-hello.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'curses'
include Curses

init_screen                     # スクリーンを初期化する

begin
  setpos(4, 9)                  # 5行目の10桁目(0から数えるため)
  addstr("こんにちは")
  refresh                       # 出力を画面に反映させる
  sleep 3                       # これがないとすぐ消えてしまうため
ensure
  close_screen
end
```

プログラム中で利用している setpos、addstr、refresh メソッドは以下のような働きを持つ。

setpos(*y*, *x*)	端末画面上 *y*+1 行目、*x*+1 桁目にカーソル位置を設定する。
addstr(*string*)	端末画面上の現在のカーソル位置に文字列 *string* を表示する。
refresh	それまでの出力を実際に端末画面に反映させる。

9.5 即時キー入力

これまで作成してきたようなデータの入力を主目的とするプログラムでは、意味のある文字列を打ち込んで最後に **Return** キーを押して初めてデータがプログラム（gets メソッド）に渡る。データ入力では間違えずに入れることが大切なのでこれでよいが、たとえば「準備ができたら何かキーを押してください」、「y か n を押してください」のような場合は **Return** を押さずに進めた方が利用者にしてみれば快適である。

curses ライブラリに含まれる getch メソッドは入力キーを 1 字分だけ読み取るもので、これを呼ぶ前に cbreak メソッドを呼んでおくと、**Return** を押さなくてもデータが getch メソッドに伝わるようになる。逆に、nocbreak メソッドを呼んでおくと **Return** を押すまで先に進まない。以下の 2 つの例を試してみよ。

リスト9.2●cur-nocbreak.rb（Returnキーを押すまで進まない）

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'curses'
include Curses

begin
  init_screen                    # スクリーンを初期化する
  nocbreak                       # Returnでデータをまとめて送る
  addstr("何かキーを押してください: ")
  x = getch                      # 1字読み取る
  addstr("\nさようなら(3秒後に終わります)")
  refresh                        # 出力を画面に反映させる
  sleep 3                        # 結果がしばらく見えるようにする
ensure
  close_screen
end
```

リスト9.3●cur-cbreak.rb（Returnキーを押さなくても進む）

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'curses'
include Curses

begin
  init_screen                    # スクリーンを初期化する
  cbreak                         # リターンキーなしでも入力させる
  noecho                         # 入力文字のエコーバックを無しにする
  addstr("何かキーを押してください(打った文字は見えません): ")
  x = getch                      # 1字読み取る
  addstr("\nさようなら(3秒後に終わります)")
  refresh                        # 出力を画面に反映させる
  sleep 3                        # 結果がしばらく見えるようにする
ensure
  close_screen
end
```

リスト 9.3 で利用した noecho は、入力した文字を画面に出すこと（エコーバック）をやめる。

9.6 制限時間付きキー入力

Curses.timeout 変数にミリ秒単位の整数を指定すると、getch メソッドで入力を待つ制限時間となる。たとえば、入力待ちを2.5秒で打ち切るにはCurses.timeout = 2500とする。

リスト9.4●cur-timeout.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'curses'
include Curses
```

```
cbreak
Curses.timeout = 2500          # 2.5秒でアウト

srand
alphabet = "abcdefghijklmnopqrstuvwxyz"
ans = alphabet[rand(alphabet.length)]

begin
  init_screen
  setpos(5,10)
  addstr(sprintf("%cを押せ! : ", ans))
  refresh
  key = getch
  setpos(6,10)
  if key == ans
    addstr("正解! また会おう")
  else
    addstr("出直してこい")
  end
  refresh
  sleep 2
ensure
  close_screen
end
```


9.7 文字属性変更

curses ライブラリのメソッドで行なう画面出力ではエスケープシーケンスなどは使えない。代わりに、curses 独自の方法で前景色・背景色・文字飾りなどの属性変更を行なう。

主な用途としては文字に色を付けることが考えられる。curses では curses 管轄下の「色番号」を使って属性を管理する。そのためには、まず自分で決めた色番号に対して前景色と背景色を組にしたものを定義してから、属性をセットするメソッドを利用する。ここでは使い勝手のよ

い attron を利用した例を示す。

リスト9.5●cur-color.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'curses'
include Curses

begin
  init_screen
  has_colors? or abort("この端末では色が使えません")
  cbreak
  start_color                     # 必ずinit_screenのあと、init_pairの前
  # init_pair(0, COLOR_BLACK, COLOR_WHITE) # 白のままだね
  init_pair(1, COLOR_RED, COLOR_BLACK)
  init_pair(2, COLOR_GREEN, COLOR_BLACK)
  init_pair(3, COLOR_YELLOW, COLOR_BLACK)
  init_pair(4, COLOR_BLUE, COLOR_BLACK)
  init_pair(5, COLOR_MAGENTA, COLOR_BLACK)
  init_pair(6, COLOR_CYAN, COLOR_BLACK)
  init_pair(7, COLOR_WHITE, COLOR_BLACK)
  init_pair(8, COLOR_BLACK, COLOR_WHITE)
  init_pair(9, COLOR_BLUE, COLOR_YELLOW)
  nc = 9
  init_pair(warn=10, COLOR_RED, COLOR_WHITE)
  # 以上定義した色番号nは color_pair(n) で利用する

  clear
  setpos(0, 0)
  1.upto(nc) do |i|
    addstr("#{i}: ")
    attron(color_pair(i)) do
      addstr("こんにちは ")
      attron(A_BOLD) do
        addstr("太こんにちは ")
      end
      attron(A_REVERSE) do
        addstr("逆こんにちは ")
      end
      attron(A_REVERSE|A_BOLD) do
```

```
          addstr("逆太こんにちは ")
        end
      end
      addstr("\n")
    end
    refresh
    addstr("何かキーを押すと終了します.\n")
    refresh
    getch
  ensure
    close_screen
  end
```

attron による属性設定は、色や属性を表すビット値の合成値を指定する。Ruby によるビット演算子一覧を示しておく。

表9.1●Rubyのビット演算子

ビット演算子	働き
m & *n*	ビット AND
m | *n*	ビット OR
m ^ *n*	ビット XOR
~*n*	ビット反転
m << *n*	*m* の *n* ビット左シフト
m >> *n*	*m* の *n* ビット右シフト

9.8 サブウィンドウ

curses が統轄する画面は Curses::Window というクラスのオブジェクトで、最初のウィンドウは stdscr 変数に割り当てられたオブジェクトである。これまで説明したメソッドは、すべて stdscr に対する処理を行なうためのものである。

cursesでは、`stdscr`とは別の新たなウィンドウを生成してそのウィンドウだけを処理対象とすることができる。プルダウンメニューや会話的出力の箱型ウィンドウなどは、サブウィンドウを用いて作成する。

サブウィンドウはsubwinメソッドで生成する。`Curses::Window`オブジェクトを返すのでその値を保存しておき、サブウィンドウ内への出力を行ないたい場合にそれを利用する。

- `win.subwin(lines, cols, y, x)`
  ウィンドウ*win*の内部に*lines*行×*cols*桁のサブウィンドウを作成する。サブウィンドウの左上の位置は*win*内の*y*+1行、*x*+1桁目とする。

- `win.box(vch, hch)`
  ウィンドウ*win*の最外郭に枠を付ける。枠左右の縦線に使う文字は*vch*、枠上下の横線に使う文字は*hch*にする。

たとえば、端末画面の10行目5桁の位置に10行×30桁のサブウィンドウを作り枠を作るには以下のようにする。

リスト9.6●cur-box.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'curses'
include Curses

begin
  init_screen
  w = stdscr.subwin(5, 36, 10, 5)       # サブウィンドウを作成し w に格納
  w.box("|"[0], "-"[0])                 # サブウィンドウに枠付け
  w.setpos(2, 10)                       # サブウィンドウ内の位置指定
  w.addstr("キーを押すと終了")            # サブウィンドウに表示
  w.refresh
  w.getch
ensure
  close_screen
end
```

これにより以下のようなウィンドウが出現する。

図9.1●サブウィンドウの表示例

サブウィンドウを作成した場合、親ウィンドウとの境目を明確にするため、上の例のようにboxメソッドによって枠を付けると見やすくなる。ただし、枠付きのサブウィンドウに大量の文字列を表示したい場合は、枠を描く文字が上書きされないよう、内側にさらに小さいサブウィンドウを作成してそこに文字列を表示させるとよい。

サブウィンドウを含めたウィンドウには、それぞれ独立した挙動の属性を設定することができる。たとえば、ウィンドウ内部の出力文字列をスクロールさせるかはscrollokメソッドによって設定できる。また、上の例示プログラムにあるように、refreshメソッド、getchメソッドもサブウィンドウ固有のものが使える。さらに、サブウィンドウごとにgetchのタイムアウト値も設定できる。

これらのことを利用し、2つの枠付ウィンドウを作成して一方の内部にスクロール可能なサブウィンドウ、もう一方の内部にスクロールしないサブウィンドウを作成して表示実験を行なうプログラムを示す。

リスト9.7●cur-subwin.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'curses'
include Curses
```

```
def winbox(win, scroll, tcolor) # 枠を付け文を出し続けるメソッド
  win.box("|"[0], "-"[0])       # まず枠を付ける
  # 次に子ウィンドウを枠の内部に作成
  subw = stdscr.subwin(win.maxy-2, win.maxx-2, win.begy+1, win.begx+1)
  subw.scrollok(scroll)         # スクロールの許可を決める
  win.setpos(0, 2)              # 枠を上書きする位置にタイトル表示
  win.addstr(sprintf("[スクロール %s]", scroll.inspect))
  win.refresh                   # winに対する変更を反映
  # 以後は内部の子ウィンドウに対する処理のみ
  subw.setpos(0, 0)             # 子ウィンドウの左上隅に移動しておく
  i = 0
  while true                    # 枠内に「n行目\n」を永遠に出し続ける
    subw.attron(color_pair(tcolor)) do
      subw.addstr(sprintf("%2d行目\n", i+=1))
    end
    subw.refresh
    # refresh                   # このrefreshの有無でカーソル位置が違う
    sleep 0.1
  end
end

begin
  init_screen
  has_colors? or abort("この端末では色が使えません。")
  cols>=72 && lines>=20 or abort("端末の大きさ72x20以上で起動してください。")

  start_color
  init_pair(1, COLOR_RED, COLOR_WHITE)    # 1=白地に赤
  init_pair(2, COLOR_YELLOW, COLOR_BLACK) # 2=黒地に黄

  leftwin = stdscr.subwin(10, 30, 10, 05) # 左側子ウィンドウ作成
  rightwin = stdscr.subwin(10, 30, 5, 40) # 右側子ウィンドウ作成
  t1 = Thread.new do                      # 左窓を操作するスレッド生成
    winbox(leftwin, true, 1)              # 第2引数がスクロール指定
  end
  t2 = Thread.new do                      # 右窓を操作するスレッド生成
    winbox(rightwin, false, 2)            # 第2引数がスクロール指定
  end
  setpos(0, 0)                            # 画面先頭にメッセージ出力
  attron(color_pair(2)) {
```

```
    addstr("何かキーを押すと止まります")
  }
  refresh
  getch                                 # 1字読み捨てて終了
  # t1.kill; t2.kill
ensure
  close_screen
end
```

この例では左右それぞれの枠付ウィンドウ内部で文字列を連続して表示させるため、2 つのスレッドを生成して同時実行させている。実行してしばらく経過した様子を以下に示す。

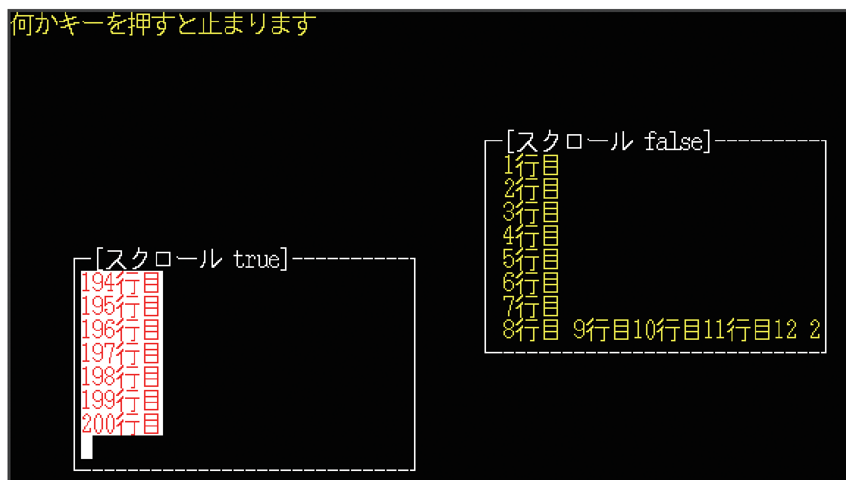


図9.2●2つのウィンドウを表示する例

この例の左側のウィンドウは自動スクロールが有効化されているため次々と行が出力されているが、右側では無効化されているため 8 行目以降はどんどん外れていっている。

9.9 キーパッド

キーボードで英数字や記号の割り当てられているキーをタイプすると、端末上ではその1字に対応するASCIIコードがプログラムに送られる。しかし、矢印キーやファンクションキーなどの特殊キーはそれに対応するASCII文字があるわけではなく、標準ではESC文字から始まるエスケープシーケンスをまとめて入力したのと同じことになる。端末上で**C-v**をタイプしてから上矢印キーを押すと、どのような文字列が生成されるかが分かる。

C-v ↑
（^[[Aと出てくる）

先頭の^[はこの記号2つでESC文字1字を表す表記で、実際にカーソルを移動してみると^[の部分が分割できない1字になっていることが分かる。端末上で特殊キーに標準的に割り当てられている文字列は以下のとおりである（^[はESC文字）。

表9.2●端末上で特殊キーに標準的に割り当てられている文字列

キー	割り当てられている文字列
↑	^[[A
↓	^[[B
→	^[[C
←	^[[D
F1	^[[11~
F2	^[[12~
:	:
F12	^[[24~

プログラム中で利用者に矢印キーやファンクションキーを使わせたい場合、それらが複数文字を送り込んで来ることを想定するのは大変なので、cursesでは特殊キーをまとめて1個のシンボルとしてgetchできるモードが用意されている。

特殊キー読み込みを利用する場合は、利用したいウィンドウオブジェクトに属するkeypadメソッドを呼ぶ。たとえば、メインウィンドウ（stdscr）で読み取るgetchで特殊キーを使

いたい場合は次のようにする。

```
stdscr.keypad(true)
```

この場合、getch で特殊キーを押したときに返る値は以下のようなシンボルで定義されている値となる。

表9.3●getchで特殊キーを押したときに返る値

キー	getch で返る値
↑	KEY_UP
↓	KEY_DOWN
→	KEY_RIGHT
←	KEY_LEFT

どのような特殊キーがどんな値を返すかは、おおむね /usr/include/curses.h ファイル内で define されている KEY_ で始まるシンボルを見ると分かる。ただし、すべてのキーが読み取り可能とは限らない。keypad(true) とする場合でも、特殊キーが入力しやすい場所にあるとは限らないので ASCII コードを持つ普通のキーにも同じ機能を持たせるように設計すべきである。たとえば、上下左右を**↑**、**↓**、**←**、**→**キーで操作する機能を付けるならば、同等のキー割り当てを **C-p**、**C-n**、**C-b**、**C-f** や **k**、**j**、**h**、**l** にも割り当てる。前者は Emacs の、後者は vi の標準キー割り当てなので説明なしでもとっさに使いこなしてもらえる可能性が高い。

矢印キーでマークを移動できるプログラムの例を cur-keypad.rb に示す。

リスト9.8●cur-keypad.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'curses'
include Curses

begin
  init_screen
  x = cols/2
  y = lines-2
  noecho
```

```
  stdscr.keypad=true            # ここをコメントアウトしてみよ
  setpos(y, x)                  # 最初の1個を書いておく
  addstr("●")
  setpos(0, 0)
  addstr("←→かhlで左右，F9かcで中央に戻す。F10かqで終了")

  refresh
  while true
    c = getch
    setpos(y, x)
    addstr("  ")                # 以前書いた丸を消す
    case c
    when KEY_LEFT, ?h
      x -= 1
    when KEY_RIGHT, ?l
      x += 1
    when KEY_F9, ?c
      x = cols/2
    when ?q, KEY_F10, ?\n
      break
    else
      setpos(1, 0)
      addstr(sprintf("不明なキー: %s", c.inspect))
    end
    x = 1 if x < 1
    x = cols-2 if x > cols-2
    setpos(y, x)
    addstr("●")
    setpos(1, 0)                # 邪魔なのでカーソルをどかす
    refresh
  end
ensure
  close_screen
end
```

9.10 その他必要なメソッド

curses ライブラリを用いた対話的なプログラムを作るために有用なメソッドや変数のうち主要なものを示す。

表9.4●cursesライブラリの主要なメソッド

メソッド	説明
cbreak	Return キーなしで即入力データを渡す。
nocbreak	Return キー入力を待ってから入力データを渡す。
echo	タイプした文字を画面にエコーバックする。
noecho	タイプした文字を画面にエコーバックしない。
clear	画面全体を表すウィンドウをクリアする。
closed?	すでに curses 画面制御が終了したかを返す。
cols	画面に表示可能な桁数を返す。
lines	画面に表示可能な行数を返す。cols とともに利用して、想定する処理を行なうのに必要な画面サイズがあるかあらかじめ確認しておくのが望ましい。（使用例、リスト 9.9）
doupdate	refresh よりも効率的に画面更新を行なう。
getstr	文字列を読み込みその値を返す。
delch	カーソル位置の 1 バイト分の文字を削除して詰める。（使用例、リスト 9.10）
insch(*ch*)	カーソル位置の 1 バイト分の文字 *ch* を挿入する。（使用例、リスト 9.10）
deleteln	カーソル位置の行を削除し 1 行分詰める。（使用例、リスト 9.11）
insertln	カーソル位置に 1 行空行を追加。（使用例、リスト 9.11）
maxx、maxy、curx、cury	それぞれその Curses::Window オブジェクトの桁数、行数、現在のカーソル位置のカラム番号、現在のカーソル位置の行番号、を返す。
inch	そのウィンドウの現在のカーソル位置に表示されている文字（1 バイト）の文字コードを返す。

リスト9.9●linesの使用例

```
require 'curses'
include Curses

if lines < 25 then
```

```
  STDERR.puts("端末を25行以上にしてから起動してください")
  exit 1
elsif cols < 80
  STDERR.puts("端末を桁幅80桁以上にしてから起動してください")
  exit 1
end
```

リスト9.10●delchとinschの使用例（cur-insdel.rb）

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'curses'
include Curses

init_screen
x, y = cols/4, lines/2
cbreak
noecho
setpos(y-3, 0)
addstr("タイプした文字を逆向きに挿入します。\n")
addstr("(Returnで終了，C-h(BS)で後方削除)")
setpos(y, x)
refresh

begin
  while (ch=getch).ord != ?\n.ord
    # .ord は文字コードを得るメソッド。
    # Ruby1.8ではgetchは入力文字の文字コードを返す一方，
    # Ruby1.9ではgetchは長さ1の文字列を返すのだが，
    # 制御文字だけは文字コードで返すため，両方強制的に文字コードに
    # 変換してから比較するようにしないと1.8と1.9で動かない。
    setpos(y-1, x)
    addstr(sprintf("key = %s", ch.inspect))
    setpos(y, x)
    if ch.ord == 0x8.ord        # C-h
      delch
    else
      insch(ch)
    end
```

```
    refresh
  end
ensure
  close_screen
end

deleteln
```

リスト9.11●deletelnとinsertlnの使用例（cur-line.rb）

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'curses'
include Curses

begin
  init_screen
  y = lines/3
  cbreak
  noecho
  setpos(y, 0)
  addstr("キーを押すと終了")
  refresh

  Thread.new do
    getch                        # キー入力があったら
    exit                         # exitするだけのスレッド
  end
  while true
    0.upto(9) do |i|
      setpos(y, 0)
      insertln                   # 1行挿入してから
      addstr("行番号 #{i}")      # 文字列出力
      sleep 0.2
      refresh
    end
    setpos(y, 0)
    0.upto(9) do |i|
      deleteln                   # y行目を1行削除
```

```
      refresh
      sleep 0.2
    end
  end
ensure
  close_screen
end
```

9.11 サンプルプログラム

curses を用いたアクション型プログラムの例を示す。

リスト9.12●cur-jump.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
# cursesを用いて●を動かす
require 'curses'
include Curses
noecho                              # エコーバックなし
cbreak                              # Returnなしで即入力
Curses.timeout = 0                  # 入力は待たない

init_screen                         # 画面も消える
ball = "●"
kesu = "  "
wait = 0.03                         # タイマー

x = 0
y = lines-2                         # 下から2行目
j = 0                               # ジャンプの高さ
jmax = 6                            # 2ステップ分高度をあげる
jnow = 0                            # 現在のステップ(0～3)
```

```
setpos(1, 0)
addstr("SPCでジャンプ!")
setpos(y-5, cols/2+rand(3))          # ランダムに決めた位置に
addstr("★")                          # ★を置く
begin
  h = y-1                            # 高さの初期値をセットしておく
  while x < cols                     # 右から左へ
    setpos(h, x-1)                   # カーソルを今の位置へ
    addstr(kesu)                     # 前のボールを消す
    x += 1
    h = y-1 - ((jmax-jnow)*jnow/2)   # ジャンプは2次曲線
    setpos(h, x-1)                   # カーソルを次の位置へ
    addstr(ball)                     # ボールを書く
    setpos(0,0)                              # カーソルを邪魔でないところへ
    refresh                          # これをしないと画面に反映されない
    if jnow > 0 then
      jnow -= 1                      # ジャンプ中の処理
      getch                          # ジャンプ中に押されたキーは捨てる
    else
      key = getch
      if key == " "[0]               # SPCだったら
        jnow = jmax                  # ジャンプ開始
      end
    end
    sleep(wait)                              # 一定時間休む
  end
  setpos(y-1, 0)
  addstr("おしまい\n")
  refresh                            # 最後も忘れずに
  sleep 3
ensure
  close_screen
end
```

実際に動かすと、●記号が左から右に向かって進み、スペースキーを押すとジャンプする。スペースキーを押した直後の画面の様子を次に示す。

図9.3●サンプルプログラムの実行結果

練習問題

9.1 練習問題 4.1 の `clock.rb` と同等のものを、curses を利用して作り直したプログラム `cur-clock.rb` 作成せよ。ただし、値の入力を行なう位置は、最初の入力が画面の最下行から数えて 3 行目、2 つ目の値の入力が最下行から 2 行目であるとする。
［ヒント］複数のカーソル位置を管理する場合、curses ではサブウィンドウを利用して、`stdscr` とサブウィンドウで個別にカーソル位置設定（`setpos`）すればよい。

9.2 後出しじゃんけんゲーム `jan.rb` を作成せよ。以下のような挙動とする。

1. 起動後、画面にグー、チョキ、パーいずれかを表示し、それを見たユーザが勝負手を入力するのを一定時間だけ待つ。
2. ユーザの勝負手の入力は次のようにする。
   グー　　←または **g**
   チョキ　↑または **c**
   パー　　→または **p**
3. ユーザが 3 連勝したらステージクリアとする。
4. 第 1 ステージの勝負手入力待ち時間は 2 秒で、同様に第 2、第 3 ステージはそれぞれ 1 秒、0.5 秒とする。
5. 第 3 ステージをクリアしたらユーザを讃えて終了、そうでなければゲームオーバー表示で終了する。

実行時の画面例を次に示す。

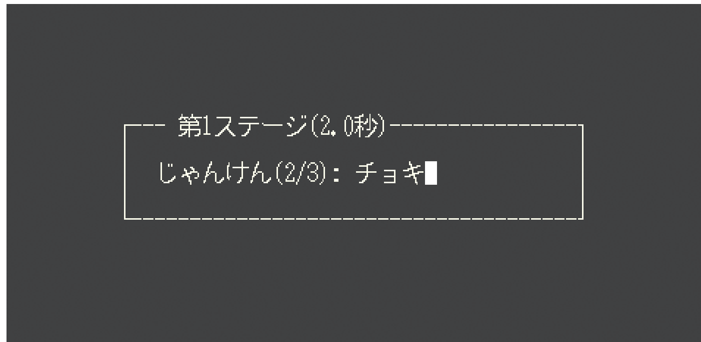


図9.4●実行時の画面の例

9.3 英単語インベーダゲーム wordvador.rb を作成せよ。以下のような挙動とする。

1. 起動すると画面右端から 1 つの英単語がゆっくりと左に向かって進んで来る。
2. プレイヤーは英単語の最初の文字から順にタイプする。
3. タイプ文字があっていれば英単語の先頭文字が消えていく。
4. タイプ文字を間違うと、単語先頭の x 座標（カラム位置）が 80%に減らされる。
5. 最後の文字まで消したらその単語は消えてレベルアップし、そのときの x 座標がプレイヤーの得点に加算され、次の英単語で 1. から繰り返す。
6. レベルアップするごとに単語の移動速度が上がる。
7. 単語を消せずに単語が 0 カラム位置に到達したらミス。
8. ミスを 3 回するとゲームオーバー。

システムに /usr/share/dict/words ファイルがもしあれば、それを英単語の取得源にするとよい。 実行時の画面例を次に示す。

```
Level:  4  score:  199  miss: 0  w=[ogramming                    ]

                          ogramming
```

図9.5●実行時の画面

第10講

GUIプログラミングの基礎

10.1 Ruby/Tk による GUI プログラミング

これまで作成してきた対話的プログラムは、入力と出力がそれぞれ 1 つの流れでできていた。それに対して GUI（Graphical User Interface）プログラミングでは、ボタン、入力窓、メニューなど、利用者が働きかけを行なう対象が複数あり、どんな順番で働きかけが来ても対応できる形となっていなければならない。ユーザからのキーボードやポインティングデバイス（マウスなど）を用いた働きかけのことを**イベント**といい、なんらかのイベントが発生したらそれに対応してあらかじめ登録しておいたプログラム部分が動くような構成となっている。このような動きを取るプログラムを**イベント駆動型プログラム**といい、GUI プログラムの典型的な形式である。

GUI プログラムでは、ウィンドウ部品の作成やイベント処理はライブラリを用いて行なう。GUI 用の部品が一式揃ったライブラリのことを**ツールキット**といい、言語や用途に応じてさまざまなツールキットが存在するが、その利用の基本的な考え方は共通している。

10.2 Ruby/Tk

GUI 用のツールキットは多種多様である。現在でも利用されているもののうち最も長い部類の歴史を持つツールキットが Tcl/Tk（http://www.tcl.tk）である。元々 Tcl/Tk は GUI を含むスクリプトを簡単に作成できることを目指して作られたスクリプト言語（Tcl）と GUI 用ツールキット（Tk）の合わさったものであるが、そのツールキット部分を他の言語から使えるようにしたものが徐々に増えた。Ruby/Tk はその Ruby 版であり、Ruby の文法で Tk を制御できる。Tk がシンプルで手軽なツールキットという地位を 1990 年代から変わらずに保ち続けていることからも、Tk の習得が比較的容易で、なおかつ今後も長期に渡って通用することが予想できる。また、基本的な概念は後発のツールキットにも共通するため、GUI プログラム入門用としても有用であると言える。

10.3 Ruby/Tk の初歩

他の GUI ツールキットを利用したプログラミングと同様、Ruby/Tk でもイベント駆動型プログラムを作成していく。そのおおざっぱな流れを次に示す。

1. ウィンドウの部品（ウィジェット）を作成する。
2. 作成した部品を土台部品に貼り付ける。
3. どのイベントにどのアクションを起こすかを登録する。
4. メインループを呼ぶ。

イベントに対する反応を特に決めないものなら 3. は不要である。まずは、ウィジェットを出すだけの簡単なプログラムを示す。

リスト10.1●tk-hello.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

TkLabel.new("text" => "Hello, world!").pack   # 1.と2.に相当
Tk.mainloop                                   # 4.に相当
```

TkLabel はラベルとなるクラスで、new によって 1 つのラベルを生成する。ラベルは、文字や画像などを表示するためのウィジェットである。引数にどのようなラベルを生成するかの情報を持たせた属性値をハッシュ形式で与えると、それに応じたラベルオブジェクトを生成する。実際には生成するだけでは表示されず、pack メソッドを用いて初めて表示される。pack は、あるウィジェットをどのようにウィンドウ上に配置するかを決めるメソッドである。このように配置を司るものを**ジオメトリマネージャ**といい、GUI 部品を効果的に配置する重要な約割を担っている。他のツールキットにも同様のものがあり、**レイアウトマネージャ**などと呼ぶこともある。

最初の例 tk-hello.rb に、イベントに対する反応を登録する部分を追加してみる。書き方

はさまざまだが、いくつかの実例を含んだ以下のプログラムを示す。

リスト10.2●tk-hello-ev.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

TkLabel.new("text" => " Hello, world! ") {
  bind('1', proc {exit})
}.pack
Tk.mainloop
```

太字で示した追加部分は TkLabel.new メソッドに与えたブロックで、このブロック内では TkLabel クラスに属すメソッド呼び出しが列挙できる。リスト 10.2 は以下のいずれの書き方でも同じ働きをする。

リスト10.3●その1:ブロックへの仮引数はそのオブジェクト自身を指す値となる。

```
TkLabel.new("text" => " Hello, world! ") {|x|
  x.bind('1', proc {exit})
  x.pack
}
Tk.mainloop
```

リスト10.4●その2:オブジェクトをローカル変数に格納してあとで使う場合。

```
lab = TkLabel.new("text" => " Hello, world! ")
lab.bind('1', proc {exit})
lab.pack
Tk.mainloop
```

リスト10.5●その3:テキスト属性指定もメソッド呼び出しで。

```
TkLabel.new() {
  text(" Hello, world! ")
  bind('1', proc {exit})
  pack
}
Tk.mainloop
```

リスト10.6●その4:packがオブジェクト自身を返すのでbindメソッドが呼べる。

```
TkLabel.new() {
  text(" Hello, world! ")
}.pack.bind('1', proc {exit})
Tk.mainloop
```


10.4 イベント処理

ウィンドウ上で発生するさまざまなイベントに対して、処理を行なう部分を**イベントハンドラ**という。この登録には上述の例のとおり bind メソッドを使う。

bind メソッドは次の形式で使用する。

bind(*シーケンス*, *処理*)

*シーケンス*の指定方法を説明する前に、いくつかの指定を含む例題プログラムを示す。

リスト10.7●tk-ev.rb

```
#!/usr/local/bin/ruby
```

```
# -*- coding: utf-8 -*-
require 'tk'

def erase(widget)
  widget.value = ""                     # Entryの入力文字列を消す
end

TkLabel.new("text" => " Hello, world! ") {
  # ラベルでは，マウス第3ボタンが効き，キーは効かない
  bind('Button-3', proc {exit})         # 右ボタンがクリックされたら
  bind('Key-3', proc {exit})            # キー3が押されたら(でも効かない)
}.pack
TkEntry.new {|tke|
  # 入力窓では，マウス第3ボタン，第2ボタン，キー'q'，'x'が効く
  bind('Key-3', proc {erase(tke)})      # キー3が押されたら(これは効く)
  bind('2', proc {erase(tke)})          # 1,2,3とだけ書くとマウスボタン
  bind('Key-q', proc {                  # Key-q でも q でもよい
         erase(tke)
         Tk.callback_break              # q そのものの入力を回避
       })
  bind('x', proc {erase(tke)})          # xならキー 'x'
}.pack
puts "ラベル上のボタン3で終了"
Tk.mainloop
```

10.4.1　イベントパターンとシーケンス

シーケンスの部分には、発生するイベントにマッチするパターンを固有の記法で表したものの並びを指定する。このパターンを**イベントパターン**といい、

modifier-modifier-type-detail

の形式、あるいはその省略できる部分を省いた形式で記述する。*modifier* は修飾（モディファイア）を示すシンボルである。ここに指定できる代表的なものを表 10.1 に示す。

表10.1●イベントパターンのシンボル

シンボル	意味
`Alt`	Alt キー
`Shift`	Shift キー
`Control`	Control キー
`Lock`	CapsLock キー
`Meta` または `M`	Meta キー
`Mod1` または `M1`	ウィンドウシステムのモディファイアキー。X Window System では 5 種類のモディファイアキーが利用できる。現在どのキーがモディファイアキーとして登録されているかは、コマンドラインで xmodmap コマンドを起動してみれば分かる。
`Mod2` または `M2`	
`Mod3` または `M3`	
`Mod4` または `M4`	
`Mod5` または `M5`	
`Button1` または `B1`	マウス第 1 ボタン
`Button2` または `B2`	マウス第 2 ボタン
`Button3` または `B3`	マウス第 3 ボタン
`Button4` または `B4`	マウス第 4 ボタン
`Button5` または `B5`	マウス第 5 ボタン
`Double`	2 連
`Triple`	3 連
`Quadruple`	4 連

キー入力で複数のキーを続けて押したものを表したいときなど、複数のイベントの組み合わせをバインドすることもできる。これにはイベントシーケンスを配列化したもので表現する。たとえば、

```
bind(['Control-x', 'Control-c'], proc{exit})
```

とすると、**C-x** に続けて **C-c** を押したときの処理を定義することになる。ただしこのとき、**C-x** のみを押したときの処理が別に定義されていれば、**C-x** が押された段階でそちらも呼ばれることに注意する。

type の部分はイベントタイプで発生したイベントの種別を表すシンボルである。代表的なものを表 10.2 に示す。

表10.2●イベントの種別を表すシンボル

シンボル	意味
Button または ButtonPress	マウスボタンのクリック
ButtonRelease	押されていたマウスボタンが離された
Key または KeyPress	キーが押された
KeyRelease	押されているキーが離された
Destroy	ウィンドウが強制終了された
FucusIn	ウィンドウがフォーカスされた
FucusOut	ウィンドウフォーカスが外れた
Enter	ウィンドウ内にポインタが入った
Leave	ウィンドウからポインタが出た
Motion	ウィンドウ内でポインタが動いた

イベントシーケンス最後の部分、*detail* はイベント発生源の具体的な指定で、マウスボタンならば 1、2、3、4、5 のいずれか、キー入力ならばそのキーを表すキーシンボル（keysym）を指定する。英数字キーは文字そのものが keysym であり、たとえば r と書けば R のキーを押した場合を意味する。したがって、

```
Control-B3-Triple-Key-r
```

というイベントシーケンスは、Control キーとマウス第 3 ボタンを押しながら r のキーを 3 連打した場合のイベントを意味する。各種記号や Return キーや BS キーなどの keysym は、システムによってあらかじめ決められている。これらを調べるにはコマンドライン で、

```
% xev
```

と起動し、出てきたウィンドウ内部で調べたいキーをタイプするか、

```
% xmodmap -pke | less
```

で割り当てキーの一覧を見るとよい（X Window System の場合）。

ただし、ウィジェットによって反応できるイベントは異なり、割り当てたイベントシーケンスがどこでも効くとは限らない。たとえばラベルウィジェットではキー入力のイベントに反応

させることはできない（227 ページ、10.4.5 参照）。

10.4.2　イベントハンドラ

bind メソッドの第 2 引数には、なんらかのイベントに結び付けるイベントハンドラを指定する。ここには Ruby の Proc オブジェクト[注1]を指定する。Proc オブジェクトに渡すブロックはその位置で有効な変数が評価される。

リスト10.8●k-proc.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

TkLabel.new("text"=>"変数xが有効なブロック\nこっちでクリックするとx=5") {
  x = 5
  bind('1', proc {printf("x=%d\n", x)})
  bg("pink")
}.pack("fill"=>"x")
TkLabel.new("text"=>"変数xが有効ではないブロック\nこっちはエラー") {
  bind('1', proc {printf("x=%d\n", x)})
  bg("#aef")
}.pack

TkButton.new("text"=>"Exit", "command"=>proc{exit}).pack
Tk.mainloop
```

この例は、後者のラベルでクリックすると変数 x が未定義でエラーを起こすという分かりやすいものだが、クラス定義を用いたスクリプトではスコープによるエラーを引き起こしやすい。あえてエラーを起こすものを示す。

注 1　http://doc.ruby-lang.org/ja/2.1.0/class/Proc.html

リスト10.9●スコープによるエラーを引き起こす例

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

class Test
  def hogehoge
    puts "Hoge!"
  end
  def initialize
    @hello = "Hello"
    @path  = "pathpathpath"     # Tkで用いている変数なので上書きはよくないが...
    ohayo = "Ohayo"
    TkLabel.new("text"=>"変数確認") {
      bind("1", proc {
             printf("ohayo=%s\n", ohayo.inspect)
             printf("@hello=%s\n", @hello.inspect)
             printf("@path=%s\n", @path.inspect)
           })
    }.pack
    myself = self                     # 現在のオブジェクトをローカル変数に記録
    TkLabel.new("text"=>"メソッド確認") {
      bind("1", proc {
             p self
             hogehoge                 # エラーになり処理はここで停止
             self.hogehoge            # これもエラー
             myself.hogehoge          # これならOK
           })
    }.pack
    TkButton.new("text"=>"Exit", "command"=>proc{exit}).pack
  end
end
Test.new
Tk.mainloop
```

実際に起動して「変数確認」の部分をクリックすると以下のように出力される。

```
ohayo="Ohayo"
@hello=nil
@path=".w00000"
```

initializeメソッドで定義した変数のうち、代入どおりに出ているのはローカル変数ohayoだけである。@helloと@pathは、評価されるのがTkButtonクラス内なので、そこでの値が得られているのが分かる。本来@path変数にはTkのオブジェクトの論理的な位置を示す値が入る。

また、「メソッド確認」の部分をクリックすると以下の出力後にエラーが発生する。

```
#<Tk::Button:0x7f7ff6289d80 @path=".w00001">
```

10.4.3　command

ボタン型のウィジェットでは、bindによるイベントハンドラの登録をせずに、commandメソッドでボタンをクリックしたときの挙動を定義できる。

リスト10.10●tk-command.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

TkButton.new("text"=>"Button") {
  command(proc {puts "これはボタン"})
}.pack
TkCheckbutton.new("text"=>"Check Button") {
  command(proc {puts "これはチェックボタン"})
}.pack
TkFrame.new {|f|
  v = TkVariable.new
  TkRadiobutton.new(f, "text"=>"Radio Button") {
    command(proc {puts "ラジオボタン-1"})
    variable(v)
    value("1")
  }.pack("side"=>"left")
  TkRadiobutton.new(f, "text"=>"Radio Button") {
```

```
    command(proc {puts "ラジオボタン-2"})
    variable(v)
    value("2")
  }.pack("side"=>"left")
}.pack

Tk.mainloop
```

10.4.4　イベントハンドラへの情報

bind メソッドに指定する手続きに対し、発生したイベントの詳細情報を渡すことができる。たとえば、リスト 10.11 のようにするとクリックが起きたときの画面上のポインタの X 座標、Y 座標が得られる。

リスト10.11●tk-xy.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

TkCanvas.new {
  width(400)
  height(300)
  bind('1', proc {|x, y, a, b|
         printf("絶対:(%d,%d)\t", x, y)
         printf("相対:(%d,%d)\n", a, b)
       }, "%X %Y %x %y")
}.pack
TkButton.new() {                # なくてもいいがquitボタンを足す
  text("quit")
  command(proc{exit(0)})
}.pack("side"=>"right")

Tk.mainloop
```

このように bind メソッドの第 3 引数に空白区切りの % 置換文字列を指定すると、それらが

手続きオブジェクトに引数化されて渡される。

表10.3●%置換文字列

% 記号	意味
`%%`	% 自身
`%#`	イベントのシリアル番号
`%d`	detail フィールドの値
`%f`	Enter/Leave イベントでのフォーカス値（0 か 1）
`%k`	keycode 値
`%x`，`%y`	イベントのウィンドウ内の相対 x 座標・y 座標
`%X`，`%Y`	イベントの x 座標・y 座標
`%A`	Unicode 値
`%T`	イベントの種別
`%W`	イベントを捕捉したウィジェット

10.4.5　即時キー入力処理

文字入力を主目的としないウィジェットでは、キー入力イベントを捕捉するようにはできていない。ウィジェットの種類を問わずショートカットキーなどの即時キー入力処理を行ないたい場合は、ルートウィジェット（Tk.root）に対してキー入力を bind すればよい。

10.4.6　仮想イベント

同じ機能に複数のキー割り当てを用意したいときや、別のプラットフォーム用に異なるキー割り当てを用意したいときなどは、複数のイベントシーケンスからなる仮想イベントを作って、それに機能を割り当てるとよい。仮想イベントは TkVirtualEvent クラスのオブジェクトとして、登録したい複数のイベントシーケンスを渡して生成する。

たとえば、**C-q** のみ、**C-x C-c** の連続押し両方のキーバインドを設定したいときは以下のようにする。

```
event_quit = TkVirtualEvent.new('Control-q', ['Control-x', 'Control-c'])
# この例では2個だがイベントシーケンスは何個でも
```

```
  :
# 特定のオブジェクトのbind部分で以下のように仮想イベントを使う
 bind(event_quit, proc{exit})
```

10.5 ジオメトリマネージャ

生成した部品は土台となる部品に配置される。一つの土台には複数の部品を配置できる。このとき、それぞれの部品をどのように配置するかを決めるのがジオメトリマネージャ（レイアウトマネージャ）である。Tk では pack、grid、place のジオメトリマネージャが使える。

いずれも土台となるウィジェット 1 つに対して 1 つのジオメトリマネージャが使える（複数のマネージャを混合利用すると暴走する恐れがある）。複数のジオメトリマネージャを混在させて使いたいときは、後述するフレームウィジェット（10.6 節）を新たな土台として組み合わせる。

10.5.1 pack

pack[注2] は、最もラフに部品を配置できるジオメトリマネージャである。土台となるウィジェット（最初はルートウィジェット）の空き領域にどのように次のウィジェットを配置するかを、おおざっぱに「上の方」、「下の方」、「右の方」、「左の方」いずれかで指定して配置を決定する。

pack の使用例として、土台となる部品の上に 3 つの部品を次の順番で配置した場合の配置状態の遷移を図に示す。

1. 「部品 1」を「上の方」に配置する（`pack("side"=>"top")`）。
2. 「部品 2」を「左の方」に配置する（`pack("side"=>"left")`）。
3. 「部品 3」を「下の方」に配置する（`pack("side"=>"bottom")`）。

注 2　http://www.tcl.tk/man/tcl8.5/TkCmd/pack.htm

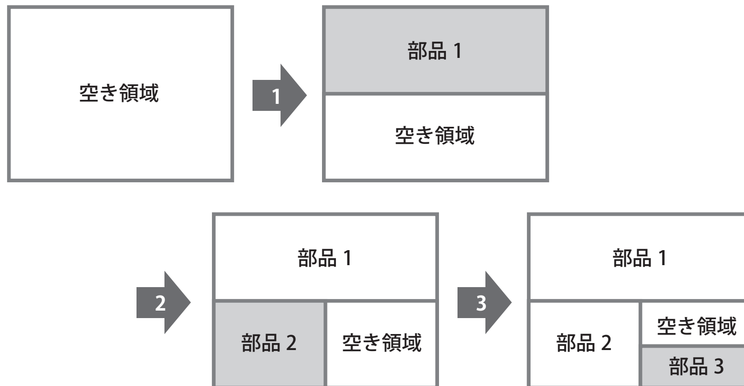


図10.1●packで3つの部品を配置していく様子

以上ですべての部品追加が完了した場合、空き領域が詰められ各部品が必要最小限の大きさに調整される。最終的な部品配置は以下のようなものとなる。

図10.2●部品の追加を完了した場合の最終的な配置

実際に上記の 3 手順で部品を配置するプログラムは次のようになる。

リスト10.12●上記の手順で部品を配置するプログラム

```
TkLabel.new("text"=>"部品1", "bg"=>"green").pack("side"=>"top")
TkLabel.new("text"=>"部品2", "bg"=>"pink").pack("side"=>"left")
TkLabel.new("text"=>"部品3", "bg"=>"yellow").pack("side"=>"bottom")
Tk.mainloop
```

　このプログラムの実行結果を次に示す。

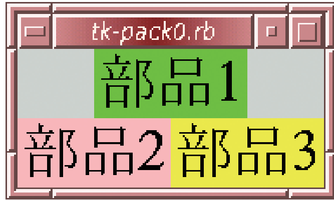


図10.3●実行結果

　「部品 1」の背景部分の範囲を見ると分かるように、部品と土台に隙間ができる場合がある。隙間を埋めたい場合は "fill" を指定する。指定できる値は次のいずれかである。

表10.4●部品配置のオプション指定

指定	説明
"x"	x 軸方向（左右両側）を埋める。
"y"	y 軸方向（上下両側）を埋める。
"both"	x・y 軸方向（上下左右）を埋める。
"none"	埋めない。

たとえば、上記の「部品 1」の貼り付けを次のように変えた場合の配置結果を図 10.4 に示す。

```
pack("side"=>"top", "fill"=>"both")
```

図10.4●部品1の配置指定に"fill"=>"both"を追加した結果

　ただし、できあがったこのウィンドウも、ウィンドウサイズを大きくすると次のようになる。

図10.5●ウィンドウサイズを大きくした様子

これは、部品 1 だけが隙間を埋める設定になっていたからで、部品 2、部品 3 も "fill"=>"both" で pack するとウィンドウサイズを大きくしたときに以下のようになる。

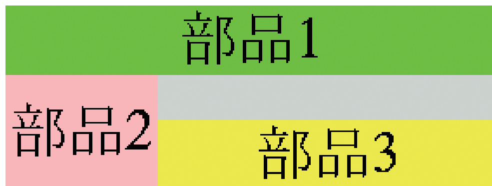


図10.6●すべての部品の配置で"fill"=>"both"を指定した結果

部品 3 の上にまだ隙間があるのは、そこが空き領域だからで、空き領域を侵蝕するように隙間を埋めさせるためには、"expand" を指定する。"expand" は、本来の持ち領域を超えてウィジェットを拡大させるかを決めるもので、これに true を設定すると有効になる。部品 3 に "expand"=>true を設定し、最終的に

```
TkLabel.new("text"=>"部品1", "bg"=>"green").pack("side"=>"top", "fill"=>"both")
TkLabel.new("text"=>"部品2", "bg"=>"pink").pack("side"=>"left", "fill"=>"both")
TkLabel.new("text"=>"部品3", "bg"=>"yellow").pack("side"=>"bottom",
                                                   "fill"=>"both", "expand"=>true)
```

として出したウィンドウを大きくすると以下のように隙間がなくなる。

図10.7●部品3の配置指定に"expand"=>trueを追加した結果

pack ジオメトリマネージャの配置を変えるための引数では、表 10.5 に示すパラメータが使える。

表10.5●配置指定で使用可能なパラメータ

パラメータ	説明
"side"	空き領域のどちら側に配置するかを、"top"（上）、"bottom"（下）、"left"（左）、"right"（右）のいずれかで指定する。
"fill"	配置するウィジェットが割り当て区画より小さいときに引き延ばして隙間を埋めるかを、"x"、"y"、"both"、"none" のいずれかで指定する。
"expand"	空き領域を埋めるように領域拡張するかを、true、false で指定する。
"before"	指定したウィジェットより前に配置する。
"after"	指定したウィジェットのあとに配置する。
"ipadx"	ウィジェットの左右の縁の内側の隙間間隔を指定する。
"ipady"	ウィジェットの上下の縁の内側の隙間間隔を指定する。
"padx"	ウィジェットの左右の縁の外側の隙間間隔を指定する。
"pady"	ウィジェットの上下の縁の外側の隙間間隔を指定する。

"ipadx"、"ipady"、"padx"、"pady" に指定するのは長さで、整数を指定するとピクセル、単位付きの整数文字列を指定するとその単位での長さになる。単位は次のいずれかで、たとえば "pady"=>"5c" のように指定する。

c　センチメートル
m　ミリメートル
i　インチ
p　ポイント（1/72 インチ）

10.5.2　grid

各部品を表形式で格子状に並べるのに適しているのが grid ジオメトリマネージャ[注3] である。

リスト10.13●tk-grid0.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'
TkLabel.new() {
  text("ラベルの1番"); bg("green")}.grid("row"=>0, "column"=>0)    # 0行0列
TkLabel.new() {
  text("ラベル2"); bg("pink")}.grid("row"=>0, "column"=>1)        # 0行1列
TkLabel.new() {
  text("ラ\nベ\nル3"); bg("pink")}.grid("row"=>1, "column"=>0)    # 1行0列
TkLabel.new() {
  text("L 4"); bg("green")}.grid("row"=>1, "column"=>1)          # 1行1列
```

このプログラムを実行すると、以下のような配置結果が得られる。

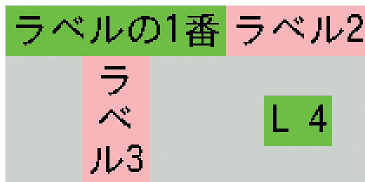


図10.8●tk-grid0.rbの配置結果

格子の各マス目の幅と高さは各列、各行が同じになるように調整される。マス目の幅・高さより小さいウィジェットは中央に配置され隙間ができる。"sticky" 属性を指定して縁に密着させる辺を、次の 1 字以上の組み合わせで指定することができる。

"n"　　上辺を密着（North）
"s"　　下辺を密着（South）
"w"　　左辺を密着（West）

注3　http://www.tcl.tk/man/tcl8.5/TkCmd/grid.htm

"e"　　右辺を密着（East）

たとえば、リスト 10.13（tk-grid0.rb）の「L4」ラベルの grid で "sticky"=>"wes" の指定を追加すると以下のようになる。

リスト10.14●tk-grid1.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

TkLabel.new() { # 0行0列
  text("ラベルの1番"); bg("green")}.grid("row"=>0, "column"=>0)
TkLabel.new() { # 0行1列
  text("ラベル2"); bg("pink")}.grid("row"=>0, "column"=>1)
TkLabel.new() { # 1行0列
  text("ラ\nベ\nル3"); bg("pink")}.grid("row"=>1, "column"=>0)
TkLabel.new() { # 1行1列
  text("L 4"); bg("green")}.grid("row"=>1, "column"=>1, "sticky"=>"wes")
Tk.mainloop
```

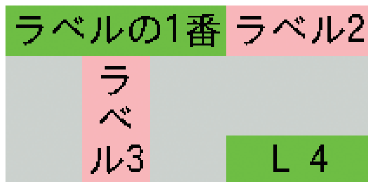


図10.9●tk-grid1.rbの実行結果

さて、余白が気になるので、すべて余白を消すことを試みる。4 つのラベルすべての grid に "sticky"=>"news" を追加すると、初期ウィンドウから余白は消える。

リスト10.15●tk-grid2.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'
```

```
TkLabel.new() { # 0行0列
  text("ラベルの1番"); bg("green")}.
  grid("row"=>0, "column"=>0, "sticky"=>"news")
TkLabel.new() { # 0行1列
  text("ラベル2"); bg("pink")}.
  grid("row"=>0, "column"=>1, "sticky"=>"news")
TkLabel.new() { # 1行0列
  text("ラ\nベ\nル3"); bg("pink")}.
  grid("row"=>1, "column"=>0, "sticky"=>"news")
TkLabel.new() { # 1行1列
  text("L 4"); bg("green")
}.grid("row"=>1, "column"=>1, "sticky"=>"nwes")

Tk.mainloop
```

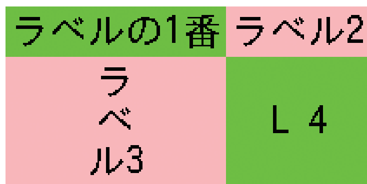


図10.10●tk-grid2.rbの実行結果

grid で作成したマス目はウィンドウサイズを変えたときにも変わらない。このため上に示したウィンドウを大きくしても周りに余白ができるだけである。

図10.11●ウィンドウサイズを大きくしたときの様子

ウィンドウサイズを変えたときに、中味のウィジェットも連動して大きさを変える設定が可

能である。これは、特定の列全体あるいは特定の行全体に対して、拡大するときの他の列・行との伸縮負担の重み付けを行なうことで制御する。上記の 4 ラベル配置例で、第 0 列と第 1 列の伸縮配分を 1:3 にするには以下の文を追加する。

```
TkGrid.columnconfigure(Tk.root, 0, "weight"=>1)
TkGrid.columnconfigure(Tk.root, 1, "weight"=>3)
```

リスト10.16●tk-grid3.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

TkLabel.new() { # 0 行 0 列
  text("ラベルの1番"); bg("green")}.
  grid("row"=>0, "column"=>0, "sticky"=>"news")
TkLabel.new() { # 0 行 1 列
  text("ラベル2"); bg("pink")}.
  grid("row"=>0, "column"=>1, "sticky"=>"news")
TkLabel.new() { # 1 行 0 列
  text("ラ\nベ\nル3"); bg("pink")}.
  grid("row"=>1, "column"=>0, "sticky"=>"news")
TkLabel.new() { # 1 行 1 列
  text("L 4"); bg("green")
}.grid("row"=>1, "column"=>1, "sticky"=>"nwes")

TkGrid.columnconfigure(Tk.root, 0, "weight"=>1)
TkGrid.columnconfigure(Tk.root, 1, "weight"=>3)
Tk.mainloop
```

第 1 引数の TkRoot は、今回 grid ジオメトリマネージャで土台となっているウィジェットで、新たな土台を作らない場合の最初の土台は Tk.Root、つまりルートウィジェットとなる。この記述を追加したウィンドウを大きくすると以下のような結果となる。

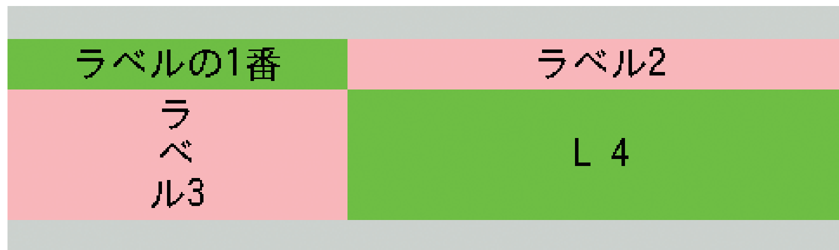


図10.12●tk-grid3.rbの実行結果（ウィンドウサイズを大きくしたときの様子）

上下に隙間があるのは行方向の設定をしていないからで、上下の隙間を埋めさせるためには TkGrid.rowconfigure で同様の設定をすればよい。

10.5.3 place

place[注4]は、ウィジェットの配置位置をx座標、y座標で直接指定できるジオメトリマネージャである。大きさの決まっている土台に座標を決めて部品を置いたり、ウィジェット間に重なりのある配置をしたい場合に有用である。

リスト10.17●tk-place.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'
TkOption.add("*font", "ipagothic 20")        # 標準フォントを大きく設定
Tk.root.width = 200
Tk.root.height= 80
TkLabel.new("text"=>"その1", "bg"=>"pink").place("x"=>10, "y"=>10)
TkLabel.new("text"=>"その2", "bg"=>"yellow").place("x"=>50, "y"=>30)
Tk.mainloop
```

注4　http://www.tcl.tk/man/tcl8.5/TkCmd/place.htm

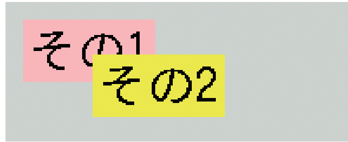


図10.13●tk-place.rbの実行結果

デフォルトでは配置するウィジェットの左上位置を基準とするが、"anchor" 属性でこれを変えることもできる。属性値には次のいずれかを指定する。

表10.6●anchor属性の属性値

属性値	説明
"n"	上辺中央
"s"	下辺中央
"w"	左辺中央
"e"	右辺中央
"nw"	左上角
"ne"	右上角
"sw"	左下角
"se"	右下角
"center"	中央

10.6 フレームウィジェット

frame[注5] は複数のウィジェットを内部に配置するためのウィジェットで、Ruby では TkFrame で生成する。

10.5.1 節で述べたとおり、1 つの土台に対しては 1 つのジオメトリマネージャしか使えない。複雑な部品レイアウトを実現するために複数のジオメトリマネージャを組み合わせたい場合は、複数のフレームをルートウィジェットに配置し、さらに各フレームごとに違うジオメトリ

注 5　http://www.tcl.tk/man/tcl8.5/TkCmd/frame.htm

マネージャを適用して内部のウィジェットを配置するようにするとよい。

たとえば次のようなレイアウトを考える。

図10.14●レイアウトの例

上半分は何かの値の入力を促すラベルとエントリを対にしたものの集合、下半分は左右に分かれたボタン。上半分については、項目名とエントリの桁位置を揃えたいので grid ジオメトリマネージャを、下半分についてはボタンを「左の方と右の方」とラフに置きたいので pack ジオメトリマネージャを使うことにする。

具体的な組み合わせ方としては、土台の上半分を占めるフレームを上から pack、残った下半分を左と右から pack、さらに上半分のフレーム内を grid で制御してラベルとボタンを配置する。

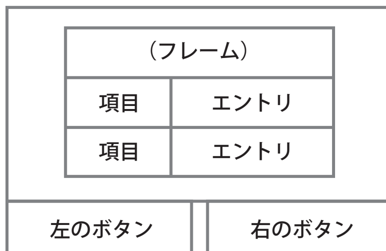


図10.15●ラベルとボタンの配置

このような配置を行なうプログラムの例を以下に示す。

リスト10.18●tk-frame.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'
```

```
=begin
+-----------------------+        +------------+
|住所      [           ] |        | j1   j2    |
|おなまえ  [       ]     |   =>   | n1   n2    |
|                       |        |            |
| [登録]         [クリア] |        | b1     b2  |
+-----------------------+        +------------+
=end

TkFrame.new {|f|
  # bg("yellow")       # レイアウトデバッグ時には背景色が有用
  j1 = TkLabel.new(f, "text"=>"住所")
  j2 = TkEntry.new(f, "width"=>20)
  n1 = TkLabel.new(f, "text"=>"おなまえ")
  n2 = TkEntry.new(f, "width"=>12)
  j1.grid("row"=>0, "column"=>0, "sticky"=>"w")
  j2.grid("row"=>0, "column"=>1, "sticky"=>"w")
  n1.grid("row"=>1, "column"=>0, "sticky"=>"w")
  n2.grid("row"=>1, "column"=>1, "sticky"=>"w")
  TkGrid.columnconfigure(f, 0, "weight"=>4) # 項目名の列
  TkGrid.columnconfigure(f, 1, "weight"=>1) # Entryの列
}.pack("fill"=>"x", "expand"=>true, "padx"=>10)
TkLabel.new("text"=>"").pack                # spacer
b1 = TkButton.new("text"=>"登録")            # 押しても何も起きない
b2 = TkButton.new("text"=>"クリア")          # 押しても何も起きない
b1.pack("side"=>"left", "padx"=>10, "pady"=>5)
b2.pack("side"=>"right", "padx"=>10, "pady"=>5)
Tk.mainloop
```

フレームウィジェットに限らず、新規のウィジェットをフレームなど別の親の子として生成するときには、ウィジェット生成の new メソッドの第 1 引数に親とするウィジェットのオブジェクトを指定する。

10.7 画像

画像を扱うには、まず元となる画像ファイルを画像オブジェクトに変換し、そののち画像を配置できるウィジェットに貼り付けるという手順をとる。

10.7.1　画像のラベルへの貼り付け

Tk の標準では gif、ppm、pgm のみ扱える。例として、`cool.gif`注6 という名前の gif 画像をラベル上に貼り付けて表示するプログラムを次に示す。画像ファイルをプログラムのファイルと同一のディレクトリにコピーしてから実行する。

リスト10.19●tk-img.rb

```
#!/usr/local/bin/ruby
require 'tk'
img = TkPhotoImage.new("file"=>"cool.gif")
TkLabel.new("image"=>img).pack
TkButton.new("text"=>"quit", "command"=>proc{exit(0)}).pack
Tk.mainloop
```

10.7.2　tkimg 拡張ライブラリ

JPG や PNG など、他の画像形式を利用する場合は、`tkextlib/tkimg/`*`FORMAT`* が必要で、たとえば、PNG 画像を使うには次の行を追加記述する。

```
require 'tkextlib/tkimg/png'
```

例として、透過部分を含む PNG 画像（`nikusoba.png`注7）を表示するものを示す。

注 6　cool.gif は http://www.yatex.org/lect/ruby/cool.gif より入手できるが、gif 画像ファイルであれば何でもよい。

注 7　nikusoba.png は http://www.yatex.org/lect/ruby/nikusoba.png より入手できるが、png 画像ファイルであれば何でもよい。

リスト10.20●tk-imgpng.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'
require 'tkextlib/tkimg/png' # PNGを利用する場合必要

img = TkPhotoImage.new("file"=>"nikusoba.png")
TkLabel.new("image"=>img, "bg"=>"pink").pack
TkButton.new("text"=>"食べる",
             "command"=>proc{puts "ごちそうさま"; exit(0)}).pack
Tk.mainloop
```

画像ファイルを作業ディレクトリに保存し、実行すると以下のようなウィンドウが現れ、[食べる] ボタンを押すと終了する。

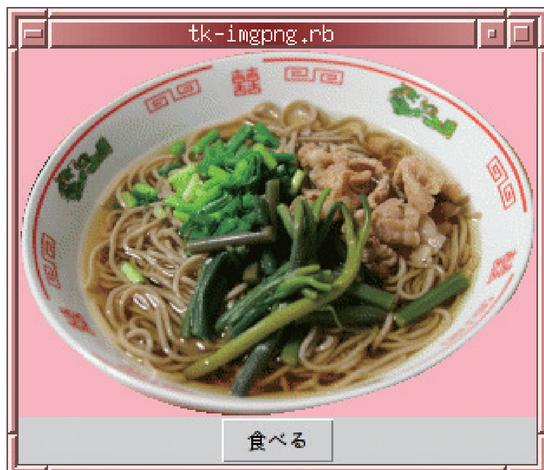


図10.16●tk-imgpng.rbの実行結果

対応している画像フォーマットは、Ruby の tkextlib ライブラリのあるディレクトリ中の tkimg/ にあるファイル一覧を見れば分かる。

```
% ls `ruby -e 'puts $:[-3]'`/tkextlib/tkimg
bmp.rb     jpeg.rb    png.rb     setup.rb   tga.rb     xbm.rb
gif.rb     pcx.rb     ppm.rb     sgi.rb     tiff.rb    xpm.rb
```

```
ico.rb      pixmap.rb  ps.rb      sun.rb     window.rb
```

上記で得られない場合は、ruby -e 'puts $:' で得られる各ディレクトリについて tkextlib/tkimg/ を探せばよい。

なお、tkimg の利用には、Ruby のライブラリだけでなくシステムに tkImg パッケージ[注8]が必要である。

10.7.3 画像の手配方法

作成したプログラムを他者に渡して利用してもらう場合、前述の 2 つの例では、プログラムそのものの他に画像ファイルも渡すか、利用者に何らかの方法で用意してもらう必要がある。その手間を利用者に掛けさせたくない場合は次に示す方法が使える。

1. ソースプログラムに埋め込んでロードする。
2. Web 経由でロードする。

以降にそれぞれの具体例を示す。

■ 1. base64 でソースプログラムに埋め込む場合

画像ファイルがあまり大きくない場合はこの方法が有効である。まず、元画像を base64 エンコードした文字列に変換する。以下のいずれかの方法で、エンコード文字列が得られることを確認する。

```
% uuencode[注9] -m image.jpg image.jpg | tail +2
% ruby -rbase64 -e 'Base64.b64encode(ARGF.read)' image.jpg
```

画像を使いたい Ruby プログラムを開き、ヒアドキュメントで base64 エンコード文字列を代入する。

注 8　http://sourceforge.net/projects/tkimg/

注 9　uuencode プログラムは、BSD 系や Solaris 系システムでは標準装備されている。Linux 系システムでは sharutils パッケージを追加インストールすることで利用できる。

```
image = <<_EOS_

_EOS_
```

のように入力しておき、挟まれた部分にエンコード文字列を挿入する。

Emacs の場合　<< の次の行にポイントを置いて **C-SPC**、**C-u**、**M-|** をタイプし、エンコードするコマンドを入力する。

vi の場合　<< の行にポイントを置いて **:**、**r**、**!** とタイプしてからエンコードするコマンドを入力する。

実際のプログラムは以下のような構成になる。

リスト10.21●tk-imgheredoc.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'
require 'tkextlib/tkimg/jpeg'

shell = <<_EOS_                # これはJPEG形式
/9j/4AAQSkZJRgABAQEASABIAAD//gAGS2FtZf/bAEMAEQwNDw0LEQ8ODxMSERUaKxwaGB
gaNSYoHys/N0JBPjc8O0VOY1RFSV5LOzxWdldeZ2pvcG9DU3qDeWyCY21va//bAEMBEhMT
GhcaMxwcM2tHPEdra2tra2tra2tra2tra2tra2tra2tra2tra2tra2tra2tra2tra2tra2
tra2tra2tra//AABEIAEQAUAMBIgACEQEDEQH/xAAaAAADAAMBAAAAAAAAAAAAAAAABQYD
BAcB/8QAMxAAAQMDAgQDBgYDAQAAAAAAAQACAwQFEQYhEjFBUWGRwQcVIjJxgRMUM2KhsS
QmQtH/xAAYAQEAAwEAAAAAAAAAAAAAAAAAAAQIDBP/EACARAQEAAgEEAwEAAAAAAAAAAAAB
AhEDEiExQQQTIlH/2gAMAwEAAhEDEQA/ALxCEIBC8cQ1pc4gAbknoud3/VlZcqp1BZy6OI
EtMjThz/HPQIL+eqp6ZvFFPPHE3u9wCXnUtlD+D3ll8nBOPm281z2LTlRUO4mr4ZzxxlxHR49CFeW
9I0bo8Gom4u+2PLCjqjT6sv46FDPFURiSCVkjDycxwIWRcsp6iv0ddBh341LJzHR49CFeW
PUNFfGyCm42SR7ujeMHHdSpZrtTdCEIgIQucajv14nv89bk2Fb5pY2xksbHFs5QZdY6Y1klq
6iSz2/8ATB4ZXjm89W/TutC1OKPozfFq4aO6aV/7xhm63Lk1XNQ0u8Wii0lp6Q
jaF622S4CUxyrYbN4qrfTbqWxVMFBPqRbzpYMVDzmgmBjcRjAlEh9vjiuM9wY+hHm6OdndQ
OxVU8zdnAYz3Nn1Vsb3Yc2P5266hY4JBNTxygkpWcKkYhP3v1XwobOaIyeZpbGZWoKOz
jmCAM+a6SoH21UzWy0NW3Z7g6M/bBH91lBO1cH5C/11KNmh7g36ZyP4WeOVb2qrfKYKC+ws
JE0MZmPMNdgYJ8DySvgbrv4PZZ5z26/j5TXSl8Vle9E3a2nGPvFHTSE3ZnGD8yTTpyPyXHT6
qsm2+VmM3TN84aWZJTMZ73WRUdDEZCCeHbPc5PkBbPc5PWRUdDEZCCeHbBPc5PIJnadMXO+ltRVONNSdpy
P2t9SrJtNatJWuSoZHwgABzju+Q9zArDm8K/nLcu08JF+ldRUFP8Am4qnKmQ+SKZ3GAO/i
```

```
otGaiku8MlLWEGqhGePGONvf6qdlvOotSzyMtrJIoBtwxHAA8Xd1S6R00+yNkqKl7X1MrQ
0hvJg54z1VmKmU7rS0TXa0tFK0OmhfxhvVwxuAqJCCF0peoLhQnT90YGngMbCduIdj2I9E
qrdHXmile2iaKqncfhLXDP3B6qk1Ho5l1qxV0crKeZ36gLdnHvt1SJti1bbCYaKaR0XQxz
DHkeSEuis2O/seGe7puI8iGAgfcbKlsWlqe0x+879JGHt+IMeQWs8T3K0P8AeKZ2P8l+R+
149Vj9x6m1BUsF0dJHEz/qXADfo0cyibbfJrX+0Gnjc5lvpHTY2D5Dwjy5/wBJMIb/AKwq
oxUAxUzd+IsLY2juO5VtadN221RNEcDZJgN5ZBlxPp9k4RDVttBBbKGOkpm4jYOZ5k9SVt
IQgEIQgEIQgEIQgEIQg//Z
_EOS_

TkLabel.new() {
  image(TkPhotoImage.new("data"=>shell))
}.pack
TkButton.new() {
  text("kick")
  command(proc {exit(0)})
}.pack
Tk.mainloop
```

■ 2. open-uri で HTTP で取得する場合

プログラムで利用する画像ファイルを、利用者が Web アクセスできる場所に置く。その URL を open-uri 拡張[注10] 込みの `open` で開き、`read` メソッド[注11] ですべて読み取った文字列を `TkPhotoImage.new` に渡す。ただし、通常行なわれる自動漢字コード変換をさせないよう、`Tk::BinaryString` メソッドに渡した結果を渡す。

リスト 10.22●tk-imghttp.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'
require 'open-uri'
require 'tkextlib/tkimg/png'

img = open("http://www.yatex.org/lect/ruby/star.png", "r") do |s|
  s.read
```

注 10 http://doc.ruby-lang.org/ja/2.1.0/library/open=2duri.html

注 11 http://doc.ruby-lang.org/ja/2.1.0/class/IO.html#I_READ

```
end

TkLabel.new() {
  image(TkPhotoImage.new("data"=>Tk::BinaryString(img)))
  bg("white")
}.pack
TkButton.new() {
  text("exit")
  command(proc {exit(0)})
}.pack
Tk.mainloop
```

10.8 フォント

文字を表示できるウィジェットでは、表示する文字のフォントを選べる。フォントはフォントオブジェクト（TkFont）で指定する。

リスト10.23●tk-font.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

txv = TkVariable.new('24のフォント')
def enlarge(me)          # meには下のラベルオブジェクトが渡されて来る
  f = me.font
  v = me.textvariable
  f.size = (f.size.to_f*1.2).to_i
  v.value = sprintf("%dのフォント", f.size)
end

f = TkFont.new("ipagothic 24 italic")
TkLabel.new() {
```

```
  textvariable txv
  text("24のフォント")
  font(f)
  bind('Button-1', proc {enlarge(self)})
}.pack
Tk.mainloop
```

フォント名の指定は、X のフォントファミリ名、サイズ、variant の 3 要素を空白で区切って指定する（後ろのものは順に省略可能）。

Tk で使えるフォントファミリの一覧は TkFont.families で得られる。

```
% irb -rtk
irb> print TkFont.families.sort.join(", ")
```

フォント名に空白が含まれる場合は、各要素を配列化するか、空白を含む文字列部分を { } で囲む。下記の 2 つは同じ指定となる。

```
f = TkFont.new(["vl gothic", 30])
f = TkFont.new("{vl gothic} 30")
```

また、フォントファミリ、サイズ、太さ、傾きを個別に 属性設定する指定方法もある。それぞれ、"family"、"size"、"weight"、"slant" で指定する。たとえば、

```
TkFont.new("mikachan 24 italic")
```

という指定は、

```
TkFont.new("family"=>"mikachan", "size"=>24, "slant"=>"italic")
```

と同様の指定に置き換えられる。

フォント指定は必ずしもフォントオブジェクトを介さず、

```
TkLabel.new("text"=>"hello", "font"=>"times 24 bold").pack
```

のようにしてもよいが、その都度フォントオブジェクトが作られ、あとから制御できないことから、共通フォントを複数のオブジェクトで 使う場合や、動的にフォントを変えたい場合はフォントオブジェクトを利用した方がよい。

```
fnt = TkFont.new("family"=>"aquafont", "size"=>20)
l = TkLabel.new("text"=>"Hello", "font"=>fnt)
```

としてラベル生成しておくと、

```
fnt.family = "y.ozfont"
fnt.size = 40
```

などとして、既存のラベルのフォントを変えられる。

10.9 代表的なウィジェット

GUI 部品を作る上で有用なウィジェットを列挙する。

10.9.1 ラベル

Tk の label[注12] に基づくラベルは表示するのみのテキストを配置することを主目的としたウィジェットで、テキストの色やフォントを変えたり、背景として画像を表示することが容易で、手軽に使える。リスト 10.1（`tk-hello.rb`）でもラベルを使用した。

ポインティングデバイス関連のイベントに反応して色を変える例を示す。

注 12 http://www.tcl.tk/man/tcl8.5/TkCmd/label.htm

リスト10.24●tk-labelev.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

TkLabel.new() {
  text("Hello, world!")
  bg("#fdd"); fg("black")
  bind("Enter", proc {bg("#dfd")})
  bind("Leave", proc {bg("#fdd")})
  bind("Button-1", proc {exit(0)})
}.pack
Tk.mainloop
```

Enterイベント（ラベル内にマウスポインタが入ること）で背景色を#dfd（淡い緑）に、Leaveイベント（同じく出ること）で背景色を#fdd（淡い赤）に変更する。

10.9.2　メッセージ

複数行に渡る文章を提示するにはmessage[注13]が使いやすい。RubyではTkMessageのオブジェクトとして生成する。パラグラフの縦横比を百分率で指定する（aspect）か、折り返し幅をピクセル数（数値）か、文字幅（文字列）で指定する（width）。

リスト10.25●tk-message.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

TkMessage.new() {
  aspect(400)
  text("これはアスペクト比400%に指定したメッセージエリアである。
ソース中の改行はそのまま改行として反映されるが，
必ずしも改行させたくない\
```

注13　http://www.tcl.tk/man/tcl8.5/TkCmd/message.htm

```
位置にはバックスラッシュを入れる。")
  bg("pink")
}.pack
TkMessage.new() {
  width(200)
  text("これは幅200ピクセルに指定したメッセージエリアである。")
  bg("yellow")
}.pack
TkMessage.new() {
  width("10c")
  text("これは幅10cmに指定したメッセージエリアである。")
  bg("#aef")
}.pack

Tk.mainloop
```

これを実行すると以下のようなウィンドウが現れる。

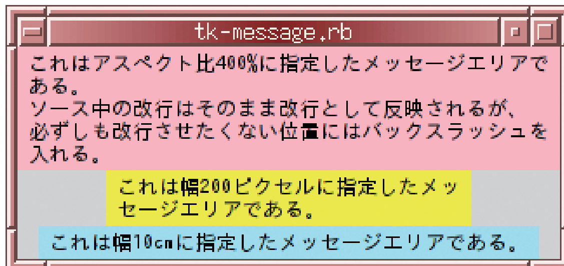


図10.17●tk-message.rbの実行結果

改行位置が柔軟に設定されていることに注意せよ。

10.9.3　ボタン

Tk の button[注14] に基づくボタンも、ラベルと同様に画像やテキストを設定できる。押したときのアクションを command メソッドで指定するのは、すでに出た例のとおりである。

注 14　http://www.tcl.tk/man/tcl8.5/TkCmd/button.htm

10.9.4　チェックボタン

チェックボタン（checkbutton[注15]）は、ボタンが押されている（ON）か解除されている（OFF）かで、2値を取得するメソッドである。Rubyでは TkCheckButton オブジェクトとして生成する。

ボタンには状態を保存するための tk 変数を割り当てて、それ経由で値を取得する。デフォルトでは OFF のとき "0" が、ON のとき "1" が得られる。tk 変数は TkVariable で生成し、それを variable メソッドで割り当てる。ただし、この tk 変数は、チェックボタンの選択・解除操作をして初めて値が入るので、チェックボタン生成時に選択（select）か、解除（deselect）しておく方がよい（例参照）。

リスト10.26●tk-checkbutton.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

v = TkVariable.new
TkCheckButton.new {
  text("抜ける")
  width("10")
  height("5")
  variable v
  deselect
}.pack
TkButton.new() {
  text(" GO! ")
  command(proc {
            if v == "1"
              puts "抜けます。"; exit 0
            else    # "0" のはず
              puts "まだまだ"
            end
          })
}.pack
Tk.mainloop
```

注15　http://www.tcl.tk/man/tcl8.5/TkCmd/checkbutton.htm

10.9.5　ラジオボタン

radiobutton[注16]は、複数のボタンでグループをなし、どれか1つだけが 選択された状態になるものである。TkCheckButton とほぼ同様の使い方だが、同じ tk 変数を使うもの同士がグループとなる。当該ボタンが押されたときに tk 変数に設定する値は value メソッドで定義しておく。

リスト10.27●tk-radiobutton.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

v = TkVariable.new
TkRadioButton.new {
  text("あじ")
  variable v
  value "鯵"
}.pack("fill"=>'x')
TkRadioButton.new {
  text("いか")
  variable v
  value "烏賊"
}.pack("fill"=>'x')
TkRadioButton.new {
  text("うなぎ")
  variable v
  value "鰻"
}.pack("fill"=>'x')
TkButton.new {
  text("決定")
  command(proc {
            if v.value == "" then     # 何も選んでいないとき
              puts("何か選んでね。")
            else
              printf("%s食べよう!\n", v.value)
              exit 0
            end
```

注16　http://www.tcl.tk/man/tcl8.5/TkCmd/radiobutton.htm

```
        })
}.pack
TkButton.new {
  text("リセット")
  command(proc {v.value = ""})
}.pack
Tk.mainloop
```


10.9.6　文字列入力

1行内の短文入力にはentry[注17]ウィジェットを利用する。RubyではTkEntryを利用する。入力窓のみのウィジェットであるため、その直前に入力ガイドを表示するlabelを配置するのが望ましい。

entryウィジェット利用時には入力された値を保持するためのtk変数を割り当て、これをTkEntryのtextvariableメソッドで指定する。このtk変数を他のウィジェットでも用いると、入力途中の値も共有される。以下の例では、名前を入力するためのentryウィジェット（e1）と、それとは全く別のlabelウィジェット（n2）でtk変数vnameを共有させている。

リスト10.28●tk-entry.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

# 入力中の値を別のウィジェットで共有させることができる。
# それには同じ TkVariable を指定する。
vname = TkVariable.new("名無")

# Entryに何を入れるべきかのラベルを付ける。
# 桁が揃った方が気持ちよいので，グリッドマネージャを使う。
TkFrame.new() {|f|
  # f はフレーム自身。内部でnewするウィジェットは親にfを指定すること
  # l1 = TkFrame.new(f) {|f2|
  #   bg("yellow")
```

注17　http://www.tcl.tk/man/tcl8.5/TkCmd/entry.htm

```
  #   TkLabel.new(f2, "text"=>"名前?:", "justify"=>"left", "bg"=>"yellow") {
  #     pack("side"=>"left", "fill"=>"both")
  #   }
  # }
  l1 = TkLabel.new(f, "text"=>"名前は?:")
  e1 = TkEntry.new(f, "bg"=>"pink", "textvariable"=>vname)
  l2 = TkLabel.new(f, "text"=>"じゅうしょは?:")
  e2 = TkEntry.new(f, "bg"=>"pink")
  TkGrid(l1, e1, "sticky"=>"news")
  TkGrid.columnconfigure(f, 0, "weight"=>1)
  TkGrid(l2, e2, "sticky"=>"e")
  TkOption.add("*foreground", "#915711")
  n1 = TkLabel.new(f) {
    textvariable vname
  }
  n2 = TkLabel.new(f) {
    text("さん こんにちは")
  }
  TkGrid(n1, n2)
  TkButton.new(f) {
    text(" 登録 ")
    command(proc {
              printf("%sにおすまいの%sさんですね!\n5万円になります。\n",
                     e2.value, e1.value)
              exit(0)
            })
  }.grid("columnspan"=>2)
}.pack("fill"=>"both", "expand"=>true)
Tk.mainloop
```

このウィンドウレイアウトは、上半分にフレームウィジェットを作成し、その中でグリッドレイアウトを利用している。実行すると以下のようなウィンドウが現れる。

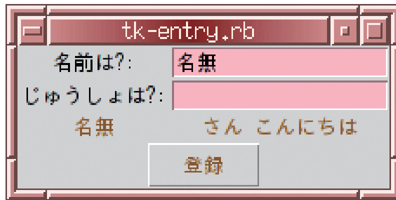


図10.18●tk-entry.rbの実行結果

10.9.7　テキストとスクロールバー

■ text ウィジェット

text ウィジェット注[18]は、短くないテキストの入力に有用で、Ruby では TkText を利用する。入力された値は value で取得する。

リスト10.29●tk-text.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

TkLabel.new("text"=>"今日の一言").pack
txt = TkText.new("width"=>40, "height"=>3).pack
TkButton.new("text"=>"登録") {
  command(proc {
            printf("復唱: %s\n", txt.value) if txt.value > ""
            exit(0)
          })
}.pack("side"=>"left", "padx"=>5, "pady"=>5)
Tk.mainloop
```

■ text ウィジェット内のマークとタグ

text ウィジェットの編集領域内には、特定の文字挿入ポイント（文字と文字の間）に好きな名前の「マーク」を付けられる。また、2 つのポイントで囲まれた領域に好きな名前の「タグ」

注 18　http://www.tcl.tk/man/tcl8.5/TkCmd/text.htm

を付けられる。タグ付けされた領域はさらに、他の部分とは独立して属性を変えることができ、たとえば特定部分だけ背景色を変えたりフォントを変えたり、あるいは bind メソッドで特定のイベントに対するアクションを定義できたりする。

以下のマークとタグは特別な意味を持つ。

insert マーク	次の入力文字が入る場所（テキストカーソルの左位置）
current マーク	マウスポインタのある場所（流動的）
sel タグ	選択された領域で次のコピーやカット操作の対象範囲

これらの値は、ユーザのカーソル移動や領域選択操作によって自動的に更新されると同時に、プログラムからも操作することができる。

■ text ウィジェット内の位置指定

text ウィジェットの編集領域内に対して文字列挿入や領域削除など、さまざまな操作を行なえる。操作対象となる場所の指定には Tk 固有の index 記法を用いる。

```
base  modifier  modifier  modifier  ...
```

という書式で、*base* は基準となる位置、*modifier* は、その位置からどれだけずらすかを指定する。たとえば、"1.0 + 3 chars" という表記は「1 行目の 0 カラム目から 3 文字後ろ」を意味し、"1.0" が *base* に、"+ 3 chars" が *modifier* に相当する。*base* として使える表記の主なものを示す。

表10.7●baseとして使える主な表記

表記	説明
line.*char*	*line* 行目の *char* 文字目。行数は 1 から、文字数は 0 から数える。*char* に end を指定すると行末の改行位置を示す。
@*x*,*y*	text ウィンドウの座標 (*x*, *y*) 位置に最も近い文字位置を示す。
end	テキスト末尾。
mark	*mark* という名前のマークの保持する位置。
tag.first	*tag* という名前のタグの保持する領域の開始位置。
tag.last	*tag* という名前のタグの領域の終了位置。

1
2
3
4
5
6
7
8
9
10

また、位置指定の *modifier* には以下のものがある。

表10.8●位置指定のmodifier

modifier	説明
+ *count* chars	基準位置から *count* 文字分後ろ
- *count* chars	基準位置から *count* 文字分前
+ *count* lines	基準位置から *count* 行分下
- *count* lines	基準位置から *count* 行分上
linestart	基準位置の行頭
lineend	基準位置の行末
wordstart	基準位置の単語先頭
wordend	基準位置の単語末

chars と lines によるずらし量はテキスト全体の先頭や末尾を越えない。また、*count* の前後の空白は省略してもよい。

■ テキスト操作メソッド

insert(*index*, *text* [, **tags*])

位置 *index* に *text* を挿入する。オプションの可変長引数 *tags* には任意個のタグ名リストを指定でき、文字列挿入と同時に挿入部分に指定したタグを割り当てる。

mark_set(*mark*, *index*)

mark という名前のマークに *index* の位置を設定する。特別なマーク "insert" に位置を設定するとカーソルがそこに移動する。

mark_unset(*mark*)

mark という名前のマークを削除する。

index(*mark*)

mark という名前のマーク位置を *line.char* の形式で取得する。

tag_add(*tag*, $index_1$, $index_2$)

tag という名前のタグを $index_1$ から $index_2$ の領域で設定する。

`tag_delete(tag)`
tag という名前のタグを削除する（複数指定可）。

`see(index)`
位置 *index* が見える位置に（必要なら）スクロールする。

`text_copy`
選択領域（sel タグ）をクリップボードにコピーする。

`text_cut`
選択領域（sel タグ）をカットしクリップボードに記憶する。

`text_paste`
クリップボードのテキストをペーストする。

その他、テキスト系ウィジェットへの操作メソッドは、http://docs.ruby-lang.org/ja/2.1.0/class/TkText.html に一覧がある。ただし、具体的な使い方の詳細は http://www.tcl.tk/man/tcl8.5/TkCmd/text.htm を併せて参照する必要がある（8.5 の部分は導入されている TclTk のバージョンに置き換える）。

■ スクロールバーの追加

限られた面積で長い文を入れさせたいときはスクロールバー（scrollbar[注19]）を付ける。スクロールバーはテキストエリアと一体化させ、ウィンドウサイズを変えても操作できるように、スクロールバーを先に pack する。

リスト10.30●tk-scroll.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

ysb = TkScrollbar.new.pack("fill"=>"y", "side"=>"right")
txt = TkText.new("width"=>40, "height"=>5, "bg"=>"#fee") {
```

注 19 http://www.tcl.tk/man/tcl8.5/TkCmd/scrollbar.htm

```
    selectbackground("pink")
    yscrollbar(ysb)
  }.pack("side"=>"right")
  TkButton.new("text"=>"決定") {
    command(proc {
              printf("[%s]\n", txt.value)
              exit(0)
            })
  }.pack("before"=>ysb, "side"=>"top")
  Tk.mainloop
```

tk-scroll.rb を実行した結果を示す。

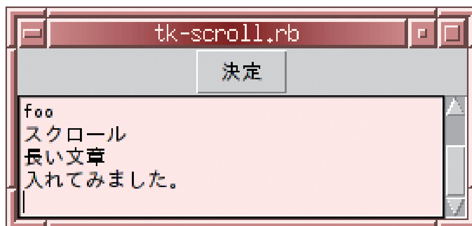


図10.19●tk-scroll.rbの実行結果

10.9.8 リストボックス

複数の候補から 1 つ、または複数の値を選ばせるときは listbox [注20] を用いる。Ruby では TkListbox のオブジェクトを生成する。

選ばせるアイテムは insert メソッドで足していく。ユーザがアイテムの選択状態を変えるたびにインスタンス変数の curselection に選んだものの添字番号が入る。 デフォルトでは 1 つのアイテムしか選べないが、selectmode を変えることにより選択操作の体系が変わる。

single	常に 1 つ選べる。
browse	常に 1 つ選べる。ボタン 1 で選択をドラッグできる。
multiple	複数選べる。ボタン 1 での選択が他の選択に影響を与えない。

注 20 http://www.tcl.tk/man/tcl8.5/TkCmd/listbox.htm

extended　複数選べる。ボタン 1 単体で押すとそれを選んで他を解除する。SHIFT を押しながらの範囲選択や、CTRL を押しながらの追加選択／解除が使える。

リスト10.31●tk-listbox.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

TkLabel.new("text"=>"何ラーメンにしますか?", "bg"=>"white").pack("fill"=>"x")
TkListbox.new {|men| # Listboxウィジェット自身が men に入る
  mode = TkLabel.new("text"=>"(1杯のみ)").pack # あとで値を変える
  insert("end", "塩")               # 末尾にアイテムを追加
  insert("end", "しょうゆ")
  insert("end", "味噌")
  insert(2, "とんこつ")              # 2番目の位置にアイテムを追加
  insert("end", "四川")
  selection_set(1)                  # 明示的に選択する
  pack("side"=>"right")
  # 以下のボタンではListboxを持つブロック変数(men)にアクセスしたいので
  # ブロック内に記述してグローバル変数化せずに済ます。
  TkButton.new("text"=>"1杯のみ") {
    command(proc{
          men.selectmode = "single"        # 1つだけ選べる
          mode.text("(1杯のみ)")            # 連動してラベルを変える
          # 選ばれたものの添字の配列が curselection に入っている
          men.curselection[1..-1].each do |i|
            men.selection_clear(i)         # 明示的に選択解除
          end
         })
  }.pack("fill"=>"x")           # デフォルトは "side"=>"top"
  TkButton.new("text"=>"何種類も") {
    command(proc{
          men.selectmode = "extended"      # 何個でも選べる
          mode.text("(何種類も)")           # 連動してラベルを変える
         })
  }.pack("fill"=>"x")
  TkButton.new("text"=>"決定") {
    bg("#efe")
    command(proc{
          for i in men.curselection
```

```
                printf("%sラーメン一丁\n", men.get(i))
              end
            })
  }.pack("fill"=>"x")
}
TkButton.new("text"=>"店を出る") {
  bg("#ecc");  command(proc{exit})
}.pack("side"=>"bottom", "fill"=>"x")

Tk.mainloop
```

tk-listbox.rb を実行して、[何種類も] ボタンを押して selectmode を "extended" にしてから CTRL+ クリックで複数の候補を選択している様子を示す。

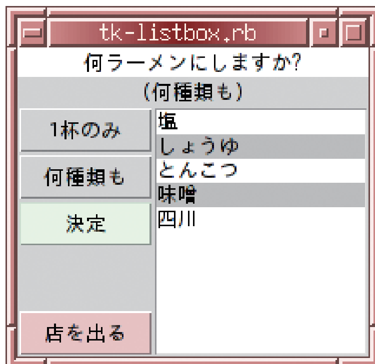


図10.20●tk-listbox.rbの実行例

アイテムが多いときは、スクロールバーを付けることもできる。100 個のダミーアイテムを生成したリストボックスにスクロールバーを付ける例とその実行結果を次に示す。

リスト10.32●tk-listboxscr.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

TkListbox.new() {
  yscrollbar(TkScrollbar.new.pack("fill"=>"y", "side"=>"right"))
```

```
  0.upto(100) do |i| insert("end", i.to_s+"番のアイテム") end
}.pack("side"=>"right")
TkButton.new("text"=>"quit", "command"=>"exit").pack
Tk.mainloop
```

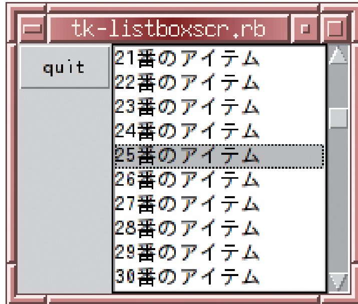


図10.21●tk-listboxscr.rbの実行結果

10.9.9　スケール

一定範囲の整数を選ばせるためには数直線状のスケールを出す scale[注21] を用いる。Ruby では TkScale を利用する。

1 から 12 までの整数を数直線状のスケールで選べるプログラムの例を示す。

リスト10.33●tk-scale.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

v = TkVariable.new
TkScale.new() {
  variable v
  from  1
  to    12
  set   Time.now.month
```

注 21　http://www.tcl.tk/man/tcl8.5/TkCmd/scale.htm

```
    # showvalue false
    label "開始月"
    orient "horizontal"        # or "vertical"
    command(proc {STDERR.printf("\r%d", v)})
  }.pack
  # 同じtk変数で連動するウィジェットを作れる(なくてもよい)
  TkEntry.new("textvariable"=>v).pack
  TkButton.new("text"=>"Set") {
    command(proc {printf("\n%s", `cal #{v} #{Time.now.year}`); exit(0)})
  }.pack
  Tk.mainloop
```

このプログラムでは、入力値を保持する tk 変数 v を TkEntry ウィジェットと共有させ、スケール部分のスライドでも、入力窓への直接数値入力どちらでも 数値指定ができるようになっている。さらに、TkScale ウィジェットで値が変更されたときの処理を command で設定(STDERR.printf の部分)しているため、端末画面にも選択途中の数値がその都度出力される。

図10.22●tk-scale.rbの実行結果

10.9.10 スピンボックス

spinbox[注22] は、指定した範囲の数値をエントリボックスで直接入力させつつ、マウスクリックでも数値の増減を制御できる（下図参照）。

図10.23●スピンボックス

Ruby では TkSpinbox で作成する。

リスト10.34●tk-spinbox.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

v = TkVariable.new
TkSpinbox.new() {
  textvariable v                # tk変数を利用するならtextvariableで
  to    12                      # toを先に指定する必要がある。
  from  1
  font  "times 18"
  # values [1,3,5,7,8,10,12]    # 有効な値を配列で与えることも可
  set   Time.now.month
  width 4                       # 入力窓の幅
  bg    "khaki"
  command(proc {STDERR.printf("\r%d", v)})
}.pack
TkButton.new("text"=>"Set") {
  command(proc {printf("\n%s",
          `cal #{v} #{Time.now.year}`); exit(0)})
}.pack
Tk.mainloop
```

注 22 http://www.tcl.tk/man/tcl8.5/TkCmd/spinbox.htm

scaleウィジェット同様、commandによって値が変更されたときの動きを定義できる。

10.9.11　メニュー

GUIアプリケーションのためのメニューはmenu[注23]ウィジェットで作成する。Rubyからは、一括でメニューバーを作れるTkMenubarクラスを用いると手軽に構築できる。

たとえば、以下のようなメニュー構成を作るものとする。

「Fファイル」
　「O開く」
　「C閉じる」
　「--」
　「Q終了」
「E編集」
　「Cコピー」
　「Xカット」
　「Vペースト」

このメニュー階層を表す配列をTkMenubarに与えて以下のようにする。

リスト10.35●tk-menubar.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

menuspec=
  [[["F ファイル", 0],
    ["O 開く", proc {puts "open"}, 0],
    ["C 閉じる", proc {puts "close"}, 0],
    ["--"],
    ["Q 終了", "exit", 0]],
   [["E 編集", 0],
```

注23　http://www.tcl.tk/man/tcl8.5/TkCmd/menu.htm

```
    ["C コピー", proc {puts "copy"}, 0],
    ["X カット", proc {puts "cut"}, 0],
    ["V ペースト", proc {puts "paste"}, 0]]]

TkFrame.new {|f|
  pack("fill"=>"x")
  TkMenubar.new(f, menuspec).pack("side"=>"left")
}
TkScrollbar.new {|s|
  pack("side"=>"right", "fill"=>"y")
  TkText.new("width"=>40, "height"=>10, "bg"=>"#f8f0f0") {
    yscrollbar(s)
  }.pack("side"=>"right")
}
Tk.mainloop
```

menuspec の値にある整数 0 はアクセラレータ指定で、たとえば、

```
["Preference", proc {setpref()}, 7]
```

とすると、メニュー文字列 "Preference" の 0 から数えて 7 バイト目、つまり n に下線が引かれ、キーボードで n をタイプしたときにその項目が選ばれるようになる。日本語メニューを作りたい場合でもメニュー文字列を日本語だけにせず、アクセラレータキーを先頭に書いておくとよい。

ただし、TkMenubar によるメニューではカスケードメニュー（メニューの 1 アイテムを選ぶとさらにメニューが出てくるもの）は作れない。その他、メニューのアイテムにはチェックボタンや ラジオボタンも作れるが、これらは TkMenu や TkMenubutton を直接制御する必要がある。メニューのみ作成するサンプルプログラムを示す。

リスト10.36●tk-menu.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

TkOption.add("*font", "ipagothic 20")
```

```
menubar = TkFrame.new {
  relief       "raised"
  borderwidth  3
}.pack("fill"=>"x")

menu_f = TkMenubutton.new(menubar) {
  text "File"
  underline 0
}.pack("side"=>"left")
menu_e = TkMenubutton.new(menubar) {
  text "Edit"
  underline 0
}.pack("side"=>"left")

filemenu = TkMenu.new(menu_f, "title"=>"ファイルメニュー")
filemenu.add('command',
             "label"=>"O 開く",
             "command"=>proc {puts "open"},
             "underline"=>0)
filemenu.add('command',
             "label"=>"C 閉じる",
             "command"=>proc {puts "close"},
             "underline"=>0)
filemenu.add('separator')
filemenu.add('command',
             "label"=>"Q 終了",
             "command"=>"exit",
             "underline"=>0)

editmenu = TkMenu.new(menu_e, "title"=>"編集メニュー")
editmenu.add('command',
             "label"=>"C コピー",
             "background"=>"pink",
             "command"=>proc{puts "copy"},
             "underline"=>0)
editmenu.add('command',
             "label"=>"X カット",
             "command"=>proc{puts "cut"},
             # "columnbreak"=>1,        # 次の行に行く
             "underline"=>0)
editmenu.add('command',
```

```
            "label"=>"V ペースト",
            "command"=>proc{puts "paste"},
            "underline"=>0)

zoom = TkVariable.new("100")
zoommenu = TkMenu.new(menu_e)

zoommenu.add("radiobutton",
            "label"=>"50%",
            "variable"=>zoom,
            "value"=>"50",
            "command"=>proc{
              puts zoom.value
              zoom.value="50"
            },
            "underline"=>0,
            "indicatoron"=>true)
zoommenu.add("radiobutton",
            "label"=>"100%",
            "variable"=>zoom,
            "value"=>"100",
            "underline"=>0,
            "indicatoron"=>true)
zoommenu.add("radiobutton",
            "label"=>"200%",
            "variable"=>zoom,
            "value"=>"200",
            "underline"=>0,
            "indicatoron"=>true)

zoommenu.add("command",
            "label"=>"see zoom",
            "underline"=>0,
            "command"=>proc {STDERR.puts zoom.value})
editmenu.add('cascade',
            "label"=>"Z ズーム",
            "menu"=>zoommenu,
            "underline"=>0)


menu_f.menu(filemenu)
```

```
  menu_e.menu(editmenu)

  TkScrollbar.new {|s|
    pack("side"=>"right", "fill"=>"y")
    TkText.new("width"=>40, "height"=>10, "bg"=>"#f8f0f0") {|t|
      yscrollbar(s)
      # 第3ボタンクリックでzoommenuを出す
      bind('Button-3', proc{|x, y| zoommenu.popup(x, y)}, "%X %Y")
    }.pack("side"=>"right")
  }
  Tk.mainloop
```

このプログラムでは次のような階層メニューが出る（実際に機能するのは「ファイル→終了」のみ）。

図10.24●tk-menu.rbの実行結果

10.9.12　ダイアログ

ユーザになんらかの明示的な確認をさせたいときに、新たな小ウィンドウを出してメッセージとともに確認ボタンを押させるものが messageBox[注24] である。Ruby では `Tk.messageBox` メソッドで作成する（クラスではなくメソッド）。独立したウィンドウを作成するため既存ウィ

注 24　http://www.tcl.tk/man/tcl8.5/TkCmd/messageBox.htm

ジェットに pack したりする必要はない。

ダイアログウィンドウの作成は次の形で行なう。

```
Tk.messageBox(ハッシュ)
```

ハッシュには以下のキーと値の組み合わせが指定できる。

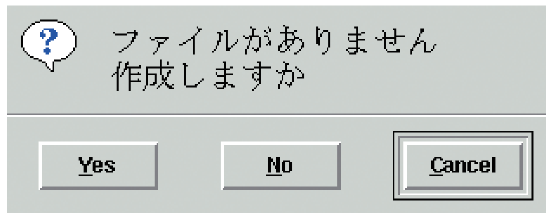
表10.9●ダイアログボックス作成のオプション

<table>
<tr><th>キー</th><th>説明</th></tr>
<tr><td>default</td><td>デフォルトで選択されるボタン。</td></tr>
<tr><td>icon</td><td>アイコンの種類。"error"、"info"（デフォルト値）、"question"、"warning" のいずれか。</td></tr>
<tr><td>message</td><td>出力するメッセージ文字列。</td></tr>
<tr><td>parent</td><td>親ウィンドウ（その上に出現する）。</td></tr>
<tr><td>title</td><td>ウィンドウタイトルとする文字列。</td></tr>
<tr><td>type</td><td>提示されるボタンセットのタイプ。
<table>
<tr><th>値</th><th>説明</th></tr>
<tr><td>abortretryignore</td><td>[abort]、[retry]、[ignore] の 3 つのボタン</td></tr>
<tr><td>ok</td><td>[ok] ボタンのみ</td></tr>
<tr><td>okcancel</td><td>[ok] ボタンと [cancel] ボタン</td></tr>
<tr><td>retrycancel</td><td>[retry] と [cancel] ボタン</td></tr>
<tr><td>yesno</td><td>[yes] と [no] ボタン</td></tr>
<tr><td>yesnocancel</td><td>[yes]、[no]、[cancel] の 3 つのボタン</td></tr>
</table>
</td></tr>
</table>

このメソッドを呼ぶと、選択されたボタンの名前の文字列が返る。たとえば、

```
Tk.messageBox('type'=>'yesnocancel',
'default'=>'cancel',
'message'=>"ファイルがありません\n作成しますか",
"icon"=>"question")
```

とすると、



のようなウィンドウが出され、そのまま **Return** を押すとデフォルトの Cancel が選ばれ、メソッドの返却値として "cancel" が返る。別のボタンを押すとそれに対応する値が全部小文字の文字列として返る。

10.9.13　Canvas

丸や多角形、直線などの部品だけでなく、他のウィジェットの土台ともなりうる多機能なウィジェットが canvas[注25] である。

Canvas 内に配置できるそれ専用のウィジェットが各種揃っている。それらは、Tkc で始まる名前のもので、new のときの第 1 引数に親となる canvas ウィジェット、残りの引数に座標や大きさなど、図形の形に則した値を指定して生成する。一度生成した図形は自動的に親となる Canvas に貼り付けられ、生成した後でも属性を変更することができる。たとえば、canvas 内の座標や大きさを決める属性値を後から変更し、画面上でアニメーションのように動かしたりするなどのことが容易にできる。

リスト10.37●tk-canvas.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

w = 400
c = TkCanvas.new("width"=>w, "height"=>w).pack          # 正方形のCanvas
TkButton.new("text"=>'quit', 'command'=>'exit').pack    # 終了ボタン
r = 100                                                 # 円の初期半径=100
width = c.width
cx, cy = c.width/2, c.height/2
```

注 25　http://www.tcl.tk/man/tcl8.5/TkCmd/canvas.htm

```
# 円は，外接する四角形の対角頂点座標を4値で指定して生成する
ovl = TkcOval.new(c, cx-r, cx-r, cx+r, cy+r,
                  'fill'=>'yellow', 'outline'=>'black')

Thread.new {                 # Tk.mainloopをメインスレッドで走らすため
  while true                 # この無限ループは別スレッドで動かす
    r = (r+8)%(width/2)      # rを8ずつ増やして枠を超えたら戻るように
    # あとで(楕)円のパラメータを変更できる
    ovl.coords(cx-r, cx-r, cx+r, cy+r)
    sleep 0.1
  end
}
Tk.mainloop
```

この例では、0.1 秒ごとに円の面積を変えている。この部分は Thread を生成して別スレッドで実行しているが、ツールキット付属のタイマである TkAfter を利用する方がよい。

```
TkAfter.new(ミリ秒, 繰り返し回数, 処理)
```

の形式でタイマオブジェクトを生成し、以下のメソッドで制御する。

表10.10●タイマオブジェクトの制御メソッド

メソッド	説明
start	開始する
stop	停める
cancel	停める
skip	処理を 1 回飛ばす
restart	再開する
continue	再開する（待ち時間を再設定可）
info	情報配列を返す

第 1 引数の実行間隔は整数で与えることもできるが、手続オブジェクトを渡して、その返却値で各回の待ち時間を動的に変えることもできる。第 2 引数に -1 を指定すると処理を実行し続ける[注26]。

注 26（参考 URL）http://jp.rubyist.net/svn/rurema/doctree/trunk/refm/api/tk_off/tkafter.rd.off

リスト10.38●弾むボール（tk-ball.rb）

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'
require 'tkextlib/tkimg/png'

img = TkPhotoImage.new("file"=>ENV["IMG"]||"ball.png")注27
$top = img.height*2
$base = $top - img.height/2
$e = 0.8
def gety(t)
  # y = -(x-1)^2 + 1
  x = ((t-10)%20)/10
  y = -(x-1.0)**2 + 1.0
  return $base - $top*y/2
end

wait = TkVariable.new('50')
TkCanvas.new {|c|
  width img.width*3
  height $top
  cx = width/2; cy = $base
  ball = TkcImage.new(c, cx, cy, "image"=>img)
  i = 0.0
  tm = TkAfter.new(proc{val=wait.value}, -1,
                   proc {ball.coords(cx, gety(i+=1))
                   }).start
  TkScale.new {
    variable wait
    to(100)
    from(1)
    label 'wait'
  }.pack('side'=>'left')
  TkButton.new("text"=>"stop", "command"=>proc{tm.stop}).pack("side"=>"left")
  TkButton.new("text"=>"start", "command"=>proc{tm.start}).pack("side"=>"left")
  TkButton.new("text"=>"quit", "command"=>"exit").pack("side"=>"left")
}.pack
Tk.mainloop
```

注27　任意のボール画像をPNGで用意しそのファイル名を環境変数IMGに指定するか、http://www.yatex.org/lect/ruby/ball.pngを利用する。

マウスでオブジェクトをつかみ、ドラッグで移動する例を次に示す。

リスト10.39●tkc-drag.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

TkCanvas.new() {|c|
  width 400
  height 300
  TkcRectangle.new(self, 50, 50, 100, 80) {
    fill("yellow")
    ox, oy = nil, nil           # 以前の座標値
    bind("ButtonPress-1", proc {|x, y|
           ox, oy = x, y        # ボタン押された瞬間の座標を記憶
           c.cursor("fleur")    # 移動カーソルに変更
         }, "%x %y")            # %x %y はウィンドウ内の相対座標
    bind('Motion', proc {|x, y|
           next unless ox
           move(x-ox, y-oy)     # 前回からの差分だけ移動
           ox, oy = x, y
         }, "%x %y")
    bind('ButtonRelease-1', proc {
           c.cursor("")         # カーソルを戻す
           ox, oy = nil, nil}
         )
  }
}.pack
Tk.mainloop
```

10.9.14 Tkc ウィジェット

Tkc* で始まるさまざまな図形オブジェクトを示す。irb でルートウィジェットと Canvas を出し、それに順次以下のウィジェットを貼り付けていくと分かりやすい。まず irb を起動し、スレッドで Tk.mainloop を起動しておく。

```
% irb -rtk
irb> cv = TkCanvas.new('width'=>400, 'height'=>300).pack
irb> Thread.new { Tk.mainloop }
```

以下、cv の保有するキャンバスに貼り付けていく形式で例示を進める。

■ 長方形

```
TkcRectangle.new(親, x1, y1, x2, y2)
```

```
# (50,50) - (100,100) を頂点とする枠が青の長方形
r = TkcRectangle.new(cv, 50, 50, 100, 100, 'outline'=>'blue')
# 塗りつぶしは fill で
re = TkcRectangle.new(cv, 120, 170, 60, 230, 'fill'=>'green', 'outline'=>'orange')
```

枠の描き方に関して、以下の属性が設定できる。

表10.11●枠の書き方に関する属性設定

属性	説明
outline	枠の色
fill	塗りつぶしの色
width	枠の線の太さ
dash	枠の線のパターン

これらの指定は、囲みのある他の Tkc ウィジェットに対しても指定できる。枠線パターンの指定を行なう dash については、後述する折れ線（TkcLine）の項で詳しく説明する。

■ 弧

```
TkcArc.new(親, x1, y1, y2, y2, "start"=>開始度, "extent"=>角度)
```

```
# ピンクに塗りつぶした 0°～60°の扇形
a = TkcArc.new(cv, 50, 50, 100, 100, "start"=>0, "extent"=>60, "fill"=>"pink")
# 120度までに拡げてみる
a.configure('extent'=>120)
# 枠の外に移動
```

```
a.coords(50, 25, 100, 75)
# moveは相対移動
a.move(0, 25)
# スタイルを変える pieslice, chord, arc のいずれか
a.style = 'chord'
a.style = 'arc'
a.style = 'pieslice'
```

■ 画像

```
TkcBitmap.new(親, x, y, 'bitmap'=>'@XBMファイル')
TkcImage.new(親, x, y, 'image'=>イメージオブジェクト)
```

「XBM ファイル」には XBM 形式（白黒）のファイル名を、「イメージオブジェクト」には TkPhotoImage で生成したものを渡す。次の例では、ネットワークの先（HTTP）から画像を取得してくるために open-uri ライブラリを、PNG 画像を処理するために tkextlib/tkimg/png をロードする。

```
require 'open-uri'
require 'tkextlib/tkimg/png'
url = 'http://www.yatex.org/lect/ruby/star.png'
star = TkPhotoImage.new('data'=>Tk.BinaryString(open(url){|s| s.read}))

# 画像の中央が座標基準となるので扇形に重なる
img=TkcImage.new(cv, 100, 50, 'image'=>star)

# 消すには delete(他のオブジェクトも同じ)
img.delete

# 画像の左上を座標基準とするには 'anchor' 属性を 'nw' (North West)に
img=TkcImage.new(cv, 100, 50, 'image'=>star, 'anchor'=>'nw')

# TkcImageはそのままで画像を差し替える
require 'tkextlib/tkimg/jpeg'
url2='http://www.yatex.org/lect/ruby/shell.jpg'
kame = TkPhotoImage.new('data'=>Tk.BinaryString(open(url2){|s| s.read}))
img.image = kame
img.image = star
```

■ 折れ線

```
TkcLine.new(親, x1, y1, x2, y2, ....)
```

直線は折れ線用の TkcLine で描画する。繋ぎたい xy 座標を任意個指定する。

```
l = TkcLine.new(cv, 111, 80, 212, 80, 121, 145, 162, 52, 191, 142,
                111,80, 'fill'=>'red', 'width'=>3)

# 矢印は arrow 属性: first, both, last
ya = TkcLine.new(cv, 120, 200, 300, 200, 'arrow'=>'first', 'width'=>3)
ya.arrow = 'last'
ya['arrow'] = 'both'
```

矢印にする場合は、arrowshape に次の 3 つの整数値を指定して矢尻の形を決めることができる。

- 矢尻の「首」と矢の先端の距離
- 矢尻の 2 つの折り返し端に沿った線と矢の先端の距離
- 矢尻の 2 つの各折り返し端と本体線の距離

感覚的には、第 1 値を大きくすると「太い」感じの矢尻となり、第 2、第 3 値の比率で第 2 値が大きくほど鋭く、第 3 値が大きいほど開いた感じになる。

矢印にしない場合は、線の端点の形状を capstyle で変えられる。また、折れ線の頂点の鋭角側の尖らせ方を joinstyle で変えられる。

図10.25●線の端点の形状（上からbutt、round、projecting）

表10.12●線の端点の形状に関する属性設定

属性	説明
butt	角ありで長さを変えない（デフォルト値）
round	丸く角取り
projecting	角ありで太さ分だけ長く

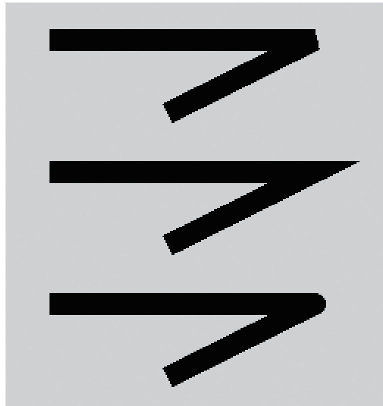


図10.26●折れ線の頂点の形状（上からbevel、miter、round）

表10.13●折れ線の頂点の形状に関する属性設定

属性	説明
bevel	尖った部分を直線で切り落とす
miter	尖らせる
round	丸くする（デフォルト値）

線のパターンを dash 属性で指定できる。 数値、あるいは数値のリストを指定できる。数値は先頭から順に線を書く長さ、書かない長さ（ピクセル値）として解釈される。値に、ピリオド（"."）、カンマ（","）、ハイフン（"-"）、アンダースコア（"_"）の各文字やその並びを指定 することもできる。ピリオドが最も細かい点の並びで、カンマ、ハイフン、アンダースコアに行くに従って点線間隔が長くなる。なお、文字列で指定する場合の点間隔は、ピクセル値ではなく線の太さに対して相対的に解釈される。たとえば、"." で指定したときは [線の太さ× 2, 線の太さ× 4] と解釈される。また、文字列の途中に空白を含めると、その分だけ線を書かない間隔が取られる。

なお、dash は線を描く他の Tkc ウィジェットでも指定できる。

■ 多角形

```
TkcPolygon.new(親, x1, y1, x2, y2, ....)
```

TkcLine と同様の座標指定で多角形を描く。終点と始点を結ぶ。

```
# シアンで塗りつぶした四角形
po = TkcPolygon.new(cv, 30, 30, 30, 160, 250, 150, 220, 35, 'fill'=>'cyan')

# 折れ線でなくなめらかに. smooth属性の true/false で決める
po.smooth = true

# smooth=trueの場合に、何分割してなめらかにするか
po.splinesteps = 2
po.splinesteps = 12

# 部品の上下関係を下に (lower)または上に (raise)
po.lower
img.raise
l.raise
```

部品間の重なりを変える raise、lower の引数に別のウィジェットを指定すると、そのウィジェットのすぐ上か下になるよう配置する。

■ 円／楕円

```
TkcOval.new(親, x1, y1, x2, y2)
```

描きたい楕円を内接する長方形の対角 2 頂点を指定する。

```
cx, cy, r = 300, 200, 30
ov = TkcOval.new(cv, cx-r, cy-r, cx+r, cy+r, 'fill'=>'navy')
# ↑は↓と同じ位置
# TkcOval.new(cv, 270, 170, 330, 230, 'fill'=>'navy')
```

■ テキストボックス

```
TkcText.new(親, x, y, 'text'=>'テキスト')
```

指定した座標にテキストを配置する。デフォルトのanchor属性は"center"でボックスの中央が配置座標の基準となる。

```
txt = TkcText.new(cv, 300, 120) {
  text 'ぐるぐる'
  fill 'darkgreen'
  font 'times 20'
}
```

中味のテキストの左右配置は、justifyにleft、right、centerのいずれかを指定して変えることができる。

上記のように、配置済みのウィジェットの属性を後から変更して、移動や変形・変色などを自由に処理できる。

```
def guru(x, y, r, ya, ov)
  theta = 2*Math::PI*Time.now.to_f
  vx = x + r*Math.cos(theta)
  vy = y + r*Math.sin(theta)
  ya.coords(120, 200, vx, vy)
  ov.coords(vx-r, vy-r, vx+r, vy+r)
end

job = TkAfter.new(50, 100, proc {guru(cx, cy, r, ya, ov)})
job.start
```

■ 座標の取り方

Canvasにウィジェットを配置する位置を決めるときは、Canvas内の相対座標を取得するリスナをバインドしておくとよい。上記のcv変数にあるCanvasであれば、

```
cv.bind('1', proc{|x, y|
        printf("(%d,%d)\n", x, y)}, "%x %y")
```

としておくと、第1ボタンのクリックで知りたい位置の座標が分かる。

10.9.15　TkcTag

TkcTagを利用すると、複数のキャンバスウィジェットをグループ化して、メンバーすべての属性を変えるといったことができる。上記の例の折れ線（l）と画像（img）をまとめて移動するには、次のようにする。

```
grp = TkcTag.new(cv)
l.tags = grp
img.tags = grp

grp.move(100, 50)
grp.move(-100, -50)

# state属性で隠す hidden, normal, disabled
grp.state = 'hidden'
grp.state = 'normal'
```

10.9.16　バウンディングボックス

単一ウィジェット、あるいはグループ化したウィジェット群をすべて包含する長方形の対角座標をbboxメソッドで取得できる。上記の例では、

```
irb> grp.bbox
=> [100, 48, 220, 150]
```

この矩形に線を引けば、ドローイングツールの矩形選択のような枠を描くことができる。

```
box = TkcRectangle.new(cv, grp.bbox, 'dash'=>'.')
```

10.9.17　Canvas内ウィジェットの検索

Canvasには子供となる多数のウィジェットを配置して使うことになるため、処理対象となる特定の子ウィジェットを選別する必要が発生する。このときに使うのがTkCanvasのfindメ

ソッドである。

```
canvas.find(Command[, Args])
```

第 1 引数 *Command* の部分には以下のいずれかが指定できる（指定時には他と区別の付く範囲で省略が可能）。

- all
  すべての子ウィジェットを返す。
- below、above
  第 2 引数にウィジェット ID、または TkcTag のタグオブジェクトを指定し、そのウィジェット（群）よりも下（below）あるいは上（above）にあるウィジェット群を配列で返す。たとえば、cv.find('below', img.id) ならば img ウィジェットより下にあるウィジェット群の配列を返す。
- closest
  canvas.find("closest", *x*, *y*) のように指定し、Canvas 内座標 (*x*, *y*) に最も近いウィジェットを返す。
- enclosed
  canvas.find("enclosed", *x1*, *y1*, *x2*, *y2*) のように利用し、第 2 〜第 5 引数で指定した座標を対角頂点とする矩形内部に含まれるウィジェット群を配列で返す。たとえば、cv.find('enc', *img.bbox) [注 28] とすると、画像ウィジェット img を囲むバウンディングボックスが検索範囲となる。
- overlapping
  上記 enclosed と同様に用い、重なりを持つウィジェット群を返す
- withtag
  次の引数に指定した TkcTag オブジェクトに含まれるウィジェット群を返す。

以上の Canvas ウィジェット群の irb 操作をまとめたものを tkc-irb.rb に示しておく。

注 28 メソッド呼び出しの引数の最後に渡す配列の前に * を付けると、配列が展開されてメソッドに渡される。たとえば、img.bbox の値が [100, 50, 220, 150] だとすると、cv.find('enclosed', *img.bbox) は、cv.find('enclosed', 100, 50, 220, 150) に開かれてメソッドが呼び出される。

リスト10.40●tkc-irb.rb

```
# -*- coding: utf-8 -*-
cv = TkCanvas.new('width'=>400, 'height'=>300).pack
Thread.new { Tk.mainloop }

r = TkcRectangle.new(cv, 50, 50, 100, 100, 'outline'=>'blue')
re = TkcRectangle.new(cv, 120, 170, 60, 230,
                      'fill'=>'green', 'outline'=>'orange')
a = TkcArc.new(cv, 50, 50, 100, 100,
               "start"=>0, "extent"=>60, "fill"=>"pink")
a.configure('extent'=>120)
a.coords(50, 25, 100, 75)
a.move(0, 25)
a.style = 'chord'
a.style = 'arc'
a.style = 'pieslice'

require 'open-uri'
require 'tkextlib/tkimg/png'
url = 'http://www.yatex.org/lect/ruby/star.png'
star = TkPhotoImage.new('data'=>
                        Tk.BinaryString(open(url){|s| s.read}))
img=TkcImage.new(cv, 100, 50, 'image'=>star)
img.delete
img=TkcImage.new(cv, 100, 50, 'image'=>star, 'anchor'=>'nw')
require 'tkextlib/tkimg/jpeg'
url2='http://www.yatex.org/lect/ruby/shell.jpg'
kame = TkPhotoImage.new('data' =>
                        Tk.BinaryString(open(url2){|s| s.read}))
img.image = kame
img.image = star
l = TkcLine.new(cv, 111, 80, 212, 80,
                121, 145, 162, 52, 191, 142,
                111, 80, 'fill'=>'red', 'width'=>3)
ya = TkcLine.new(cv, 120, 200, 300, 200,
                 'arrow'=>'first', 'width'=>3)
ya.arrow = 'last'
ya['arrow'] = 'both'
po = TkcPolygon.new(cv, 30, 30, 30, 160, 250, 150, 220, 35,
                    'fill'=>'cyan')
```

```
po.smooth = true
po.splinesteps = 2
po.splinesteps = 12
po.lower
img.raise
l.raise
cx, cy, r = 300, 200, 30
ov = TkcOval.new(cv, cx-r, cy-r, cx+r, cy+r, 'fill'=>'navy')
txt = TkcText.new(cv, 300, 120) {
  text 'ぐるぐる'
  fill 'darkgreen'
  font 'times 20'
}
def guru(x, y, r, ya, ov)
  theta = 2*Math::PI*Time.now.to_f
  vx = x + r*Math.cos(theta)
  vy = y + r*Math.sin(theta)
  ya.coords(120, 200, vx, vy)
  ov.coords(vx-r, vy-r, vx+r, vy+r)
end

job = TkAfter.new(50, 100, proc {guru(cx, cy, r, ya, ov)})
job.start
grp = TkcTag.new(cv)
l.tags = grp
img.tags = grp

grp.move(100, 50)
grp.move(-100, -50)
grp.state = 'hidden'
grp.state = 'normal'
box = TkcRectangle.new(cv, grp.bbox, 'dash'=>'.')
```

10.9.18　Canvas ウィジェット使用例

curses の例で示したキャラクタがジャンプするプログラム（リスト 9.12、cur-jump.rb）と同等のものを Ruby/Tk で作成したものを示す。

リスト10.41●tkc-jump.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'
require 'tkextlib/tkimg/jpeg'
require 'tkextlib/tkimg/png'
require 'open-uri'

class Jump
  def initialize(width = 600, height = 400)
    dir='http://www.yatex.org/lect/ruby/'
    imgsrc = {'data'=>Tk.BinaryString(open(dir+'star.png'){|s| s.read})}
    @star = TkPhotoImage.new(imgsrc)
    imgsrc['data'] = Tk.BinaryString(open(dir+'shell.png'){|s| s.read})
    @kame = TkPhotoImage.new(imgsrc)
    imgsrc['data'] = Tk.BinaryString(open(dir+'shell2.png'){|s| s.read})
    @kame2 = TkPhotoImage.new(imgsrc)
    imgsrc['data'] = Tk.BinaryString(open(dir+'shell3.png'){|s| s.read})
    @kame3 = TkPhotoImage.new(imgsrc)
    @manimg = [@kame, @kame2, @kame3]
    @getstar = nil
    @st_x, @st_y = 400, 220
    @job = nil
    f = TkFrame.new()
    @bt0 = TkButton.new(f, "text"=>"Start(TkAfter)").pack('side'=>'left')
    @bt1 = TkButton.new(f, "text"=>"Start(sleep)").pack('side'=>'left')
    @bt2 = TkButton.new(f, "text"=>"QUIT",
                        'command'=>"exit").pack('side'=>'left')
    f.pack
    # TkFrame.new {|f|
    #   @bt0 = TkButton.new(f, "text"=>"Start(TkAfter)").pack('side'=>'left')
    #   @bt1 = TkButton.new(f, "text"=>"Start(sleep)").pack('side'=>'left')
    #   @bt2 = TkButton.new(f, "text"=>"QUIT",
    #                       'command'=>"exit").pack('side'=>'left')
    # } のようにしたいところだが
    # newに渡すブロックは、TkFrameの initialize() スコープで
    # 実行されるので@bt0 などのこのクラスのクラス変数の初期化は
    # 行なえない。
    @stage = TkCanvas.new('width'=>width, 'height'=>height) {
      bind('1', proc{|x, y| printf("(%d,%d)\n", x, y)}, "%x %y")
```

```
  }.pack
  @i_x, @i_y = 20, height-20
  @g_x, @g_y = width-@kame.width/2, @i_y
  @tgt = TkcImage.new(@stage, @st_x, @st_y, 'image'=>@star)
  @man = TkcImage.new(@stage, @i_x, @i_y, 'image'=>@manimg[0])
  @unit_x, @unit_y = 10, 20
  @jmax = 6; @jnow = 0
  @wait = 20 #msec
  @tx = TkcText.new(@stage, 10, 10,
                  "anchor"=>:nw, "text"=>"Start",
                  "font"=>"Times 24")
  @tx.bind('1', proc{reset(); @move.start()})
  TkRoot.bind('space', proc{jump()})
  TkRoot.bind('q', proc{exit()})
  # TkAfterとsleep，それぞれでアニメーションを行なう
  @move = TkAfter.new(@wait, -1, proc {mv()})
  TkRoot.bind('x', proc{reset(); @move.start})
  @bt0.command = proc {reset(); @move.start}
  @bt1.command = proc {reset(); mvBySleep()}
end
def jump()
  @jnow = @jmax
end
def reset()
  @getstar = nil
  @tx.text = "Jump!"
  @x = @i_x
end
def mv()
  if @x < @g_x                # 右端に着く前は描画処理
    @y = @g_y - ((@jmax-@jnow)*@jnow/2)*@unit_y
    @man.coords(@x, @y)
    step = (@x-@i_x)/@unit_x/6%3
    @man.image = @manimg[step]
    if @stage.find('overlapping', *@man.bbox)[0] == @tgt
      @getstar = true
    end
    @x += @unit_x
    if @jnow > 0 then
      @jnow -= 1              # ジャンプ中の処理
    end
```

```
    else                           # 右端に着いたら終了
      @move.stop
      @tx.text = "You " + (@getstar ? "WIN!" : "Lose...")
    end
  end
  def mvBySleep()
    @x = @i_x
    while @x < @g_x
      @y = @g_y - ((@jmax-@jnow)*@jnow/2)*@unit_y
      @man.coords(@x, @y)
      step = (@x-@i_x)/@unit_x/6%3
      @man.image = @manimg[step]
      if @stage.find('overlapping', *@man.bbox)[0] == @tgt
        @getstar = true
      end
      @x += @unit_x
      @jnow -= 1 if @jnow > 0
      @stage.update            # canvasをupdateしないと画面に出ない!!
      sleep(@wait/1000.0)
    end
    @tx.text = "You " + (@getstar ? "WIN!" : "Lose...")
  end
end
k = Jump.new
Tk.mainloop
```

10.10 リンク集

Ruby/Tk の情報を得るために有用な URL を示す。

Ruby/Tk サンプルプログラム集

http://www.mnet.ne.jp/~tnomura/tksample.html

逆引き Ruby/Tk

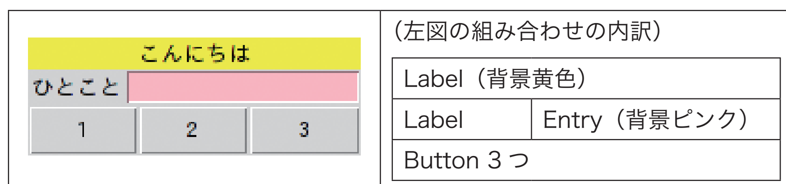
http://pub.cozmixng.org/~the-rwiki/rw-cgi.rb?cmd=view;name=%B5%D5%B0%FA%A4%ADRuby%2FTk

Ruby/Tk Guide

http://www.tutorialspoint.com/ruby/ruby_tk_guide.htm（英語だが網羅的ですべてに例が付いている）

練習問題

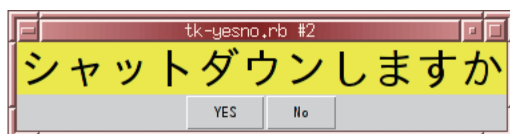
10.1 pack ジオメトリマネージャのみの組み合わせを用い、以下のようなレイアウトのウィジェット配置を行なうプログラム `tk-pack.rb` を作成せよ。



10.2 コマンドラインで第 1 引数に指定した問いかけの文字列を表示する label を上段に、YES ボタンと NO ボタンを横並びで下段に配置し、YES ボタンを押すと exit 0、NO ボタンを押すと exit 1 するようなプログラム `tk-yesno.rb` を作成せよ。たとえば、

```
% ./tk-yesno.rb シャットダウンしますか
```

のように起動すると、



のようなウィンドウが現れ、YES ／ NO のクリックに応じてプログラムは終了コード 0 ／ 1 で exit する。なお、この実行例の label フォントは、フォントサイズ 24 で `TkFont.new()` している。

10.3 前問同様、コマンドラインで第 1 引数に指定した文字列を問いかけ文として表示し、その下部に Yes、No、Cancel ボタンを配置し、それらを押すとそれぞれ exit 0、exit 1、exit -1 するようなプログラム `tk-ync.rb` を、`messageBox` メソッドを利用して作成せよ。

練習問題解答

ここに示す解答はあくまでも例の一つであり、同様の動きをするプログラムは何通りもある。

第２講

■ 2.1

cat.rb

```
#!/usr/local/bin/ruby
print ARGF.readlines.join
```

ARGF はコマンドラインに与えたファイル、または標準入力を読み取るファイルハンドルである。

■ 2.2

diary1.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'pstore'
diarydb = 'diary.pstore'

db = PStore.new(diarydb)
db.transaction do
  diary = db["diary"] = db.fetch("diary", Hash.new)
  STDERR.print "日記の日付を入力してください(YYYY-MM-DD): "
```

```
  date = gets.chomp
  STDERR.puts "文章を入力してください(終了は行頭で C-d):"
  desc = readlines.join
  diary[date] = desc
end
```

■ 2.3

diary2.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'pstore'
diarydb = 'diary.pstore'

db = PStore.new(diarydb)
db.transaction do
  diary = db["diary"] = db.fetch("diary", Hash.new)
  diary.each do |date, desc|
    printf("【%s】\n%s", date, desc)
  end
end
```

■ 2.4

ps2yaml.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'pstore'
require 'yaml/store'
diarydb = 'diary.pstore'
diaryyaml = 'diary.yaml'

db1 = PStore.new(diarydb)
db2 = YAML::Store.new(diaryyaml)

db1.transaction do
  db2.transaction do
```

```
    diary1 = db1["diary"] = db1.fetch("diary", Hash.new)
    diary2 = db2["diary"] = db2.fetch("diary", Hash.new)
    diary1.each do |key, val|
      diary2[key] = val
    end
  end
end
```


第3講

■ 3.1

mail-diary.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-

filename = Time.now.strftime("%F") + ".txt"

# ヘッダを捨てる。
while true
  break if /^$/ =~ gets
end

# 残りは本文。ファイルに保存する。
open(filename, "w") do |txt|
  txt.print readlines
end
```

dotqmail ファイルは ~/.qmail-body というファイル名でこのプログラムを起動する行を書く。

［例］ | ./path/to/mail-diary.rb

絶対パス、もしくはホームディレクトリからの相対パスで記述する。

■ 3.2

mail-uranai.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'nkf'

sender = ENV['SENDER']          # 環境変数SENDERが送信者
rcpt   = ENV['RECIPIENT']       # 環境変数RECIPIENTが受信アドレス

if sender == nil || rcpt == nil then
  STDERR.puts "$SENDER and $RECIPIENT not set. exit."
  exit 0                        # メイル用プログラムはエラーでも exit 0 すべき
elsif /.*@.*/ !~ sender then    # メイルアドレス形式でない場合
  STDERR.puts "SENDER address invalid"
  exit 0
end

kuji    = ["大吉", "吉", "中吉", "小吉", "末吉", "凶", "大凶"]
result  = kuji[rand(kuji.length)]   # 乱数でいずれかの結果を得る
to      = sender
from    = rcpt
subject = NKF.nkf("-jM", '本日の運勢')
header  = sprintf("To: %s
From: %s
Subject: %s
Content-type: text/plain; charset=iso-2022-jp
Mime-Version: 1.0\n\n", to, from, subject)
message        = NKF.nkf("-j", sprintf("今日の運勢は「%s」です。\n", result))
program        = sprintf("| sendmail -f %s %s", from, to)

open(program, "w") do |mail|
  mail.print header
  mail.print message
end
```

■ 3.3

mail-vote.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-

require 'pstore'
require 'nkf'

sender = ENV['SENDER']              # 環境変数SENDERが送信者
vote   = ENV['DEFAULT']             # -default の default に相当する部分の名前
valid  = ["foo", "bar"]             # 有効投票対象のリスト

db = PStore.new("vote.pstore")      # 投票結果格納ファイル

def sendreport(to, sj, body)        # レポート送信メソッド(投票時に利用)
  from   = ENV['RECIPIENT']         # このプログラムの受信アドレスで発信
  subj   = NKF.nkf("-jM", sj)
  header = sprintf("To: %s
From: %s
Subject: %s
Content-type: text/plain; charset=iso-2022-jp
Mime-Version: 1.0\n\n", to, from, subj)
  message = NKF.nkf("-j", body)
  program = sprintf("| sendmail -f %s %s", from, to)
  open(program, "w") do |mail|
    mail.print header
    mail.print message
  end
end

db.transaction do
  v = db['vote'] = db.fetch('vote', Hash.new)
  # 環境変数SENDERの有無で処理を切り替える
  if ENV["SENDER"] && vote then
    # メイル経由なので投票処理
    if valid.index(vote) then       # 有効投票リストにあるなら投票処理
      v[sender] = vote              # 投票者をキーとする値を設定
      sendreport(sender, "投票完了", vote+"さんに投票しました。")
    else
      v.delete(sender)              # 投票者をキーとするものを削除
```

1
2
3
4
5
6
7
8
9
10
練習問題解答

```
      sendreport(sender, "無効投票",
                 "次の候補者いずれかに投票してください: "+valid.join(", "))
    end
  else  # 環境変数設定がない場合はコマンドライン起動と見なして集計処理
    count = Hash.new(0)             # デフォルト値0のハッシュ
    v.each {|k, v| count[v] += 1}   # 候補者名のカウントを増加
    valid.each do |p|
      printf("%10s: %5d票\n", p, count[p])
    end
  end
end
```

第 4 講

■ 4.1

clock.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-

t = Thread.new {
  while true
    STDERR.printf("\e7\e[1;1H%s\e8", Time.now.to_s)
    sleep 0.1
  end
}
print("\e[2J\n")                # 画面クリア
STDERR.print "名前を入れて: "
name = gets.chomp
STDERR.print "食べたいものは: "
food = gets.chomp
```

■ 4.2

sigusr1-2.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-

class USR12
  def initialize
    @counter = 0
    printf("他の端末で kill -USR{1,2} %d\n", $$)
    Signal.trap(:USR1, proc {reset})
    Signal.trap(:USR2, proc {bye})
    while true
      STDERR.printf("%d..", @counter+=1)
      sleep 1
    end
  end
  def reset
    @counter = 0                # USR1シグナルが送られたらカウンタをリセット
    STDERR.puts "リセット!"
  end
  def bye
    STDERR.puts "さようなら"     # USR2シグナルが送られたら終了
    exit 0
  end
end

USR12.new
```

■ 4.3

rederr.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-

e = IO.pipe
pid = fork do
  STDERR.reopen(e[1])
  e[0].close                    # 子プロセスは読み込み端を閉じる
```

```
  exec(*ARGV)
end
e[1].close                          # 親プロセスは書き込み端を閉じる

Thread.new do
  while not e[0].closed?
    line = ""
    while err = e[0].gets
      STDERR.printf("\e[41m%s\e[m", err)
    end
  end
end
Process.wait
```

第5講

■ 5.1

%r の括り文字に / 以外を使えば、正規表現内に / をそのまま書ける。

```
%r,\d\d\d\d/\d\d?/\d\d?,
```

西暦部分は繰り返しを指定する { } を用いて、\d{4} としてもよい。

■ 5.2

tatekae.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'csv'

CSV.foreach("tatekae.csv") do |row|
  name, email, yen = row
  if yen && yen.to_i > 0 then
    open(email, "w") do |w|
```

```
    w.printf(<<_EOS_, name, yen.to_i)
%s 様:
日ごろより弊社にご協力を賜りありがとうございます。

さて先日の納会にお越しいただいた際にご負担いただきました
交通費(%d円)を当方負担としてお渡ししたいと思います。
次回お越しのときに印鑑をお持ちの上，受領くださいますようお願い致します。

葛戸書房 経理課
_EOS_
    end
  end
end
```


■ 5.3

email_list.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-

def email_list()
  list = open("emails.txt"){|e| e.readlines}.select {|d|
    local, domain = d.split("@")    # '@' の前後で分割
    if domain                       # @以後があるなら
      r = `host #{domain}`
      $?.to_i == 0
    end
  }.collect{|i| i.chomp}            # 行末改行も削除しておく
end

printf("Valid domains: \n%s\n", email_list.join("\n"))
```

第 6 講

■ 6.1

この問題の正解はリスト 6.11 の ft1.rb である。あるファイルが作成できるかと、ファイル名を変えることができるかは、当該ファイルではなくその置き場所となるディレクトリに書き込み権限があるかを調べることが把握できていればよい。

■ 6.2

hiscore.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'yaml/store'

mydir  = File.dirname($0)               # プログラムの存在するディレクトリ
prefix = File.expand_path("..", mydir)  # PREFIXを求める
myname = File.basename($0, ".rb")       # .rb を取り除いた名前
score  = File.expand_path("share/#{myname}.score", prefix) # スコアファイル
user   = ENV["USER"]                    # ユーザ名
rank   = nil                            # ランキング用変数を宣言しておく

answer = rand(10)                       # ゲーム部分のはじまり
STDERR.printf("入力せよ: %d\n", answer)
t0 = Time.now.to_f                      # 値入力前の時刻
while true
  break if gets.to_i == answer
end
record = Time.now.to_f - t0             # このrecordでハイスコアを決める
now    = Time.now.strftime("%F %T")     # 達成日時
thisrecord = [user, record, now]

db = YAML::Store.new(score)             # ハイスコアファイルはyamlとする
db.transaction do
  rank = db["hiscore"] = db.fetch("hiscore", Array.new) # ランキング取り出し
  rank << thisrecord                    # 今回のスコアを取り敢えず突っ込む
  rank.sort_by!{|x| x[1]}               # 各レコード先頭要素でソート
  rank = rank[0..9]                     # 最大10個にする
end
```

```
r=rank.index(thisrecord)
0.upto(rank.length-1) do |i|
  printf("%s%2d: %20s | %5.3f | %s\n", r==i ? "*" : " ", i+1, *rank[i])
end
r and printf("おめでとう! %d位にランクイン!\n", r+1)
```

このプログラム例ではハイスコアファイルに誰でも書き込めることを前提としたが、その場合は chmod a+w hiscore.score しておかなければならない。ただし、そのようにするとエディタなどでハイスコアファイルを誰でも改変できることになる。

これを防ぐため、通常のゲームプログラムでは、プログラム本体ファイルの所有者グループをゲーム用のグループ（一般的にはシステムに用意されていることが多い games グループ）にしておき、setgid 属性を付け実行時にゲーム用グループ権限で起動するようにしつつなおかつスコアファイルをゲーム用グループで書き込み可能にしておく。

しかし、Ruby プログラムのようなスクリプトはセキュリティ上 setgid が禁止されている場合がほとんどである。このような場合は、以下のような策が考えられる。

- スクリプトプログラムを起動するだけの C プログラムを作成し、その C プログラムを setgid しておく。
- sudo を用いてグループ設定して起動を許可する。

C の知識を要しない後者の方法について簡潔に説明する。Ruby インタプリタは /usr/local/bin/ruby で、hiscore.rb が /usr/local/bin にインストールしてあり、ハイスコアファイルとして /usr/local/share/hiscore.score を参照するものとする。

1. Ruby スクリプトの拡張子を除いた名前のシェルスクリプトを以下の内容で作成し、実行属性を付けておく。

```
#!/bin/sh
exec sudo -g games /usr/local/bin/ruby $0.rb
```

2. visudo で、games グループでの実行権限を与える。sudoers ファイルに以下の行を追加する。

```
ALL ALL=(:games)NOEXEC:NOPASSWD:/usr/local/bin/ruby /usr/local/bin/hiscore.rb
```

3. ハイスコアファイルを touch し、games グループ所有にしておく。

```
% sudo touch /usr/local/share/hiscore.score
% sudo chgrp games /usr/local/share/hiscore.score
% sudo chmod g+w /usr/local/share/hiscore.score
% ls -l /usr/local/share/hiscore.score
-rw-rw-r--  1 root  games    0 May 24 22:00 /usr/local/share/hiscore.score
```

以上の設定により、/usr/local/bin/hiscore と実行することにより sudo 経由で games グループ権限でゲームプログラムが実行され、ハイスコアファイルに書き込めるようになる。

■ 6.3

email_list2.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'resolv'                        # resolvライブラリ利用

def email_list()
  r = Resolv::DNS.new                   # リゾルバを初期化
  t = Resolv::DNS::Resource::IN::ANY    # レコードタイプANYで問い合わせ
  list = open("emails.txt"){|e| e.readlines}.select {|d|
    local, domain = d.chomp.split("@")  # '@' の前後で分割
    if domain                           # @以後があるなら
      begin
        x = r::getresource(domain, t)
        true                            # なくてもいいが明示的にtrueを返す
      rescue                            # 問い合わせ失敗なら
        false                           # falseを返す
      end
    end
  }.collect{|i| i.chomp}                # 行末改行も削除しておく
end

printf("Valid domains: \n%s\n", email_list.join("\n"))
```

第 7 講

■ 7.1

1.

grp.rb

```
#!/usr/local/bin/ruby
open("/etc/group", "r") do |g|
  while line = g.gets
    grp = line.split(":")
    printf("%s = %s", grp[0], grp[3])
  end
end
```

2.

```
awk -F: '{printf("%s = %s\n", $1, $4)}' /etc/group
```

3.

```
sed 's/:.*:/ = /' /etc/group
```

awk のこの挙動を Ruby で再現する -n -a -F オプションを用いると、以下のようなコマンドラインで同じ処理を行なえる。

```
% ruby -F: -na -e 'printf("%s = %s", $F[0], $F[3])' /etc/group
```

■ 7.2

例の 1 つを挙げる。

```
% cat /etc/passwd \
 | awk -F: '$3>=10000 && $3<60000{
        printf("cp -r /etc/skel/. %s && chown -R %d:%d %s\n",
                $6, $3, $4, $6)}'
```

この出力をパイプで /bin/sh に渡せば、/etc/skell ディレクトリのファイルの配布が行なえる。ただし、存在しないホームディレクトリにもコピーするので実用上はディレクトリの存在性チェックなどもすべきである。

■ 7.3

空白区切りと仮定すると 7 番目のカラムにあるため、これを抽出し、sort 後に uniq でカウントする。一例を以下に示す。

```
% grep '"GET ' access_log \
  | awk '{print $7}' | sort | uniq -c | sort -nr
```

awk では、特定の正規表現にマッチする行のみにアクションを指定できる。 以下のようにすると grep を省略できる。

```
% awk '/"GET /{print $7}' access_log \
  | sort | uniq -c | sort -nr
```

第 8 講

■ 8.1

grp.sh

```
#!/bin/sh
while IFS=: read gname ast gid users; do
  echo "$gname = $users"
done < /etc/group
```

■ 8.2

cpskel.sh

```
#!/bin/sh
while IFS=: read name ast uid gid gecos homedir shell; do
```

```
  if [ $uid -ge 10000 -a $uid -lt 60000 ]; then
    test -d $homedir && \
      cp -r /etc/skel/. $homedir && \
      chown -R ${uid}:${gid} $homedir
  fi
done < /etc/passwd
```

■ 8.3

overwrite.sh

```
#!/bin/sh
# OverWrite to file(s) after passing each contents to filter $1
#
if [ x"$1" = x"" ]; then
  cat 1>&2 <<EOF
$0 -  Overwrite wrapper for any filters
Usage: $0 "AnyFilterCommandLine" File(s)

This script WOULD NOT make backup files.
USE WITH CARE or some version control system that can
easily recover lost files.
EOF
  exit 0
fi

cmd=$1; shift
for f; do
  echo "%! $cmd
w
q" | ${EX:-ex} $f
done
```

上書きによる失敗の保証を全く行なわないので実用するには注意が必要。

第 9 講

■ 9.1

cur-clock.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'curses'
include Curses

init_screen
y = lines - 3

begin
  clockwin = stdscr.subwin(1, cols-3, 0, 0)
  t = Thread.new {
    while true
      clockwin.setpos(0, 0)
      clockwin.addstr(Time.now.to_s)
      clockwin.refresh
      stdscr.refresh
      sleep 0.1
    end
  }
  clear
  setpos(y, 0)
  addstr "名前を入れて: "
  name = getstr.chomp
  setpos(y+1, 0)
  addstr "食べたいものは: "
  food = getstr.chomp
ensure
  close_screen
end
```

■ 9.2

jan.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'curses'
include Curses

stage = 0
wait = 2000                     # /1000s
wingame = false

def shobu(win, limit)
  hands = %w,グー チョキ パー,
  win.timeout = limit
  3.times do
    win.clear
    win.setpos(1, 2)
    win.addstr("じゃんけん: ")
    win.refresh
    sleep 1
    pc = rand(3)
    hand = hands[pc]
    win.addstr(hand)
    win.refresh
    case win.getch              # グー=0, チョキ=1, パー=2 に変換する
    when KEY_LEFT, "g"[0]
      user = 0
    when KEY_UP, "c"[0]
      user = 1
    when KEY_RIGHT, "p"[0]
      user = 2
    else
      return false              # タイムアウトや他のキーは即「負け」
    end
    win.addstr(" : "+hands[user])
    win.refresh
    sleep 0.5
    if (3+pc-user)%3 != 1       # 差のmod3が1ならユーザの勝ち
      return false              # 一度でも勝ちでなかったらfalseを返して負け
    end
```

```
  end
  return true
end

begin
  init_screen
  cbreak
  noecho
  cy, cx = lines/2-3, cols/2-20
  center = stdscr.subwin(5, 40, cy, cx)
  center.box("|"[0], "-"[0])
  center.refresh
  field = center.subwin(3, 30, cy+1, cx+1)
  field.keypad(true)            # 矢印キーを使いたいので
  while (stage+=1) <= 3
    center.setpos(0, 4)
    center.addstr(sprintf(" 第%dステージ ", stage))
    center.refresh
    (wingame=shobu(field, wait)) or break
    wait /= 2
  end
  field.clear
  if wingame then
    field.addstr("\n おめでとう!!")
  else
    field.addstr("\n ざんねん...さらばじゃ")
  end
  addstr("何かのキーで終了")
  getch
ensure
  close_screen
end
```

■ 9.3

wordvador.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'curses'
```

```
include Curses

# 単語は予めシャッフルした配列にしておく
w = open(ENV["DICT"] || "/usr/share/dict/words").readlines.shuffle
miss = 0                        # ミスの回数
wait = 1.0                      # 単語が1桁部分進む間隔の基準値
level = 1                       # このレベルでwait値を割る
score = 0                       # プレイヤーのスコア
word = nil                      # 攻撃対象の英単語
begin
  init_screen
  x, y = 0, 2
  cbreak
  noecho
  stage = stdscr.subwin(1, cols, y, 0)
  stth = Thread.new {           # キー入力用のスレッド
    while true
      case stage.getch
      when word[0]
        word = word[1..-1]      # 先頭1字を削る
      else
        x = x*8/10              # 間違えたら80%に減らす
      end
    end
  }
  while miss < 3
    clear
    word = w.shift.chomp
    x = cols-word.length-2
    while x > 0                 # 単語表示桁が先頭になるまで繰り返す
      setpos(0,0)
      addstr(sprintf("Level:%3d  score:%5d  miss: %d  w=[%-32s]",
                     level, score, miss, word))
      stage.clear
      stage.setpos(0, x)
      stage.addstr(word)
      stage.setpos(0,0)
      stdscr.refresh
      stage.refresh
      if word == ""             # すべての文字を撃破したら
        level += 1              # レベルUPして次の単語へ
```

```
        break
      end
      sleep wait/level
      x -= 1
    end
    x <= 0 and miss+=1          # 先頭桁に到達したらミスカウントを増やす
    score += x
  end
  stth.kill                     # キー入力スレッドを停める
  clear
  setpos(lines/2, 0)
  addstr(sprintf("You've got %d points at level %d.\n", score, level))
  addstr("Hit enter to leave:")
  getstr
ensure
  close_screen
end
```

以下のような改良の余地がある。

- 得点がウィンドウの広さに依存するので，ステージの広さを統一する。
- 単語の長さを一定範囲にする。
- ハイスコアを登録する。

第 10 講

■ 10.1

大まかに、上下 3 段で考える。

1 段目　「こんにちは」の Label

2 段目　Label と Entry を包含する Frame

3 段目　残り領域を左から pack する 3 つの Button

3 段目の Button が隙間を埋めるように expand と fill 属性を設定する。

tk-pack.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

label = TkLabel.new("text"=>"こんにちは", "bg"=>"yellow").pack("fill"=>"both")
TkFrame.new {|f|
  TkLabel.new(f, "text"=>"ひとこと").pack("side"=>"left")
  TkEntry.new(f, "bg"=>"pink").pack("side"=>"left")
}.pack
TkFrame.new() {|f|
  background "pink"
  TkButton.new(f, "text"=>" 1 ").pack("side"=>"left", "expand"=>true, "fill"=>"both")
  TkButton.new(f, "text"=>" 2 ").pack("side"=>"left", "expand"=>true, "fill"=>"both")
  TkButton.new(f, "text"=>" 3 ").pack("side"=>"left", "expand"=>true, "fill"=>"both")
}.pack("side"=>"top", "fill"=>"both", "expand"=>true)

Tk.mainloop
```

■ 10.2

tk-yesno.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

message  = ARGV[0] || "よろしいですか"
TkLabel.new("text"=>message  , "bg"=>"yellow") {
  font(TkFont.new("size"=>24))
}.pack("side"=>"top", "expand"=>true, "fill"=>"both")
TkFrame.new() {|f|
  background "pink"
  TkButton.new(f, "text"=>" YES ") {
    command proc {exit 0}
  }.pack("side"=>"left", "fill"=>"both")
  TkButton.new(f, "text"=>" No ") {
    command proc {exit 1}
  }.pack("side"=>"left", "fill"=>"both")
```

```
}.pack("side"=>"top")
Tk.mainloop
```

■ 10.3

tk-ync.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'tk'

message  = ARGV[0] || "よろしいですか"
case Tk.messageBox("icon" => "question",
                   "message" => message,
                   "type" => "yesnocancel")
when "yes"
  exit 0
when "no"
  exit 1
when "cancel"
  exit -1
end
```

このようなプログラムを作成しておくと、シェルスクリプトを GUI 化できる。呼ぶ側のシェルスクリプトは以下のような構成にすればよい。

```
#!/bin/sh

tk-ync.rb "プログラムを終了します。
設定を保存しますか?"
code=$?
if [ $code -eq 0 ]; then
  # yesの場合の処理
elif [ $code -eq 1 ]; then
  # noの場合の処理
else        # exit -1の場合 $? は 255
  # cancelの場合の処理
fi
```

あとがき

案内役としての本書の役割はここまでである。このあともっと技術を高めていくには、複数の分野のより深い知識を得る必要がある。これから先、どのようなキーワードの知識を学習すればよいかを簡単にまとめておく。

第 2 講に関連した、データの取り扱いについて次に学ぶことは、「データ構造」と「アルゴリズム」がキーワードとなる。また、データ格納については近代のデータベースの基盤となっているリレーショナルデータベースについて入門書まででよいので目を通すと視野が広まる。簡単に使い始められる RDBMS の SQLite3 はそのよい学習土台となるだろう。

第 3 講ではメイル配送の処理について触れたが、HTTP など他のサービスについてもある程度通信規約を知ればスクリプティングの題材となる。Ruby には種々の通信規約を実現したライブラリが多数あるので、マニュアルのライブラリ一覧を眺め、見付かったライブラリ名でさらに調査を進めると創作アイデアが湧くだろう。また、興味のあるサービスの RFC 文書を調べ、そこに書かれた情報を読み取る力を付けることもスキルアップにつながる。

第 4 講と第 6 講はシステムとの関りであるため、どんな言語を使う際も応用できる。プロセスや端末、ファイル IO の仕組などについて述べてある資料を一度は読み、できれば C 言語や OS のシステムコールについてすこしだけでも触れておくとよい。

第 7 講、第 8 講は Unix システムの入門書やシェルの解説書に繰り返し目を通すと、その都度新たな発見があり、どんどん吸収できることが期待できる。本書では見送ったが、join コマンドなどテキストファイルだけで簡単なデータベース構築ができるツールが揃っている。

第 9 講 curses については、C 言語対象に書かれた curses ライブラリの解説が役立つ。また、Python の curses 関連文書にも得るものが多い。

第 10 講 Ruby/tk の関連資料については 10.10 に記したとおりである。本格的 GUI アプリケーション作成を目指す場合は、GTK+ を選択するのが順当だが、開発が速く現行仕様に合った入門用学習資料を探すのは容易でない。学習しやすさという意味では、資料の充実している Java を GUI 基盤に選択するのも価値ある判断である。

本書に関する情報を http://www.gentei.org/~yuuji/support/sr/ に集めたので参考にして頂きたい。

最後になったが、本書を世に出すにあたって貴重な助言とともに多くの労を割いて下さったカットシステム石塚勝敏さんと武井智裕さんに深く感謝申し上げる。

索 引

■記号

! 5
#（シェルの変数展開）........ 175
##（シェルの変数展開）........ 175
#{式} 110
$$ 75
$& 19
$' 19
$(cmdline)（シェル）........ 177
$0 122
$` 19
% 6
% 記法 112
% 置換文字列（イベントハンドラ）........ 227
%（シェルの変数展開）........ 175
%%（シェルの変数展開）........ 175
%= 6
%q 112
%Q 112
%r 112
%W 112
%w 112
%x 112
&& 5
&&（シェル）........ 180
* 6
** 6
**= 6
*= 6
+ 6
+= 6
- 6
-= 6
/ 6
/= 6
:+（シェルの変数展開）........ 174
:-（シェルの変数展開）........ 174
:=（シェルの変数展開）........ 174
:?（シェルの変数展開）........ 174
< 4
<（リダイレクション）........ 27
<< 10
<<（ヒアドキュメント）........ 111
<= 5
= 6
== 4
> 5
>（リダイレクション）........ 27
>= 5
? : 5
` ` 110
|| 5
||（シェル）........ 180

■A

addstr 192
"after"（pack ジオメトリマネージャ）........ 232
ALRM（シグナル）........ 78
anchor 属性（place ジオメトリマネージャ）........ 238
and 5
ARGF 26
ARGV 26
arrowshape 277
attron 196, 197
awk 158

■B

base64 エンコード 243
bbox 281
"before"（pack ジオメトリマネージャ）........ 232
begin ～ rescue ～ end 142
bind 219
box 198
break 4
button 250

■C

Canvas 271

case 3
cbreak 193, 205
checkbutton 251
clear 205
close.write 94
close_screen 192
closed? 205
collect 10, 114
cols 205
command（ボタンウィジェット） 225
CONT（シグナル） 78
csv ライブラリ 32
curses ライブラリ 190
Curses.timeout 変数 194
Curses:Window 197
curx 205
cury 205

■ D

dbm 38
def 6
delch 205
delete 9, 11, 115
delete_if 115
deleteln 205
Dir 131
Dir.chdir 133
Dir.entries 132
Dir.getwd 133
Dir.glob 131
Dir.mkdir 134
Dir.mktmpdir 149
Dir.pwd 133
Dir.rmdir 134
doupdate 205
downto 8

■ E

each 9, 10
echo 205
egrep 155
Encoding 16
entry 253
Enumerable モジュール 113
Exception 141
exec 75
"expand"（pack ジオメトリマネージャ） 232
export 176

■ F

false 5
File クラス 120
File.basename 121
File.chmod 122
File.dirname 122
File.expand_path 120
File.link 127
File.rename 127
File.stat 127
File.symlink 127
File.unlink 124
File::LOCK_EX 55
File::LOCK_SH 55
"fill"（pack ジオメトリマネージャ） 232
find 114
flock 50
flush 100, 129
for 8
for（シェル） 182
fork 75
frame 238

■ G

getch 193
getstr 205
gif 241
grep 115
grid ジオメトリマネージャ 233

■ H

has_key? 11
head 163
HOME（環境変数） 64
HOST（環境変数） 64
HUP（シグナル） 78

■ I

if 3

if（シェル）180
IFS（シェル）185
inch 205
index 10
inode 番号 125
insch 205
insertln 205
INT（シグナル）78
IO.popen 92
"ipadx"（pack ジオメトリマネージャ）232
"ipady"（pack ジオメトリマネージャ）232

■ J

JIS コードに変換 67
JPG 241

■ K

keypad 203
keys 11
KILL（シグナル）78

■ L

length 9, 11
lines 205
listbox 259
ln コマンド 124
LOCAL（環境変数）64

■ M

maxx 205
maxy 205
mbox 形式 60
menu 265
message 249
messageBox 269
MIME エンコード 66
MTA 62
MUA 62
Mutex 85

■ N

next 4
nil 5
nkf 66, 154
nocbreak 193, 205
noecho 194, 205
not 5

■ O

open 14, 15
open-uri 245, 276
Open3 98
or 5
ord 206

■ P

pack 217
pack 228
"padx"（pack ジオメトリマネージャ）232
"pady"（pack ジオメトリマネージャ）232
pgm 241
pipe+fork+exec 99
place 237
PNG 241, 276
pos 128
ppm 241
print 27
printf 27, 35
Proc オブジェクト 223
Process モジュール 74
process.detach 91
Process.kill 78
PStone 43
puts 27

■ Q

QUIT（シグナル）78

■ R

radiobutton 252
read（シェル）184
RECIPIENT（環境変数）64
redo 4
refresh 192, 199
Regexp.new 17, 18
reject 10, 115
reject! 10
reopen 101

reverse 9
rewind 128
RFC5322 60

■ S

scale 262
scrollbar 258
scrollok 199
sed 157
seek 128
SEGV（シグナル） 78
select 10, 114
SENDER（環境変数） 64
sendmail（コマンド） 65
setpos 192
shift 9
"side"（pack ジオメトリマネージャ） 232
SIGINT 79
Signal.trap 79
SMTP 62
sort 160
sort 9
sort_by 116
spinbox 264
split 29, 30
STDERR 28
stdscr 197
step 8
"sticky"（grid ジオメトリマネージャ） 233
STOP（シグナル） 78
*str***N* 13
str.length 13
*str*1+*str*2 13
str[*B*,*L*] 13
str[*B*..*E*] 13
str[*N*] 13
stty コマンド 78
subwin 198
sync 129
synchronize（Mutex） 85

■ T

tail 164
Tcl/Tk 216
tell 128
Tempfile 147
TERM（シグナル） 78
textvariable 253
Thread 83
Thread.new 83
Thread.pass 88
times 7
tk 変数 251, 253
Tk.messageBox メソッド 269
Tk::BinaryString 245
TkAfter 272
Tkc ウィジェット 274
TkCanvas 271
TkcArc 275
TkcBitmap 276
TkCheckButton 251
TkcLine 277
TkcOval 279
TkcPolygon 279
TkcRectangle 275
TkcTag 281
TkcText 279
TkEntry 253
tkextlib 276
tkextlib 241
TkFont 246
TkFrame 238
TkGrid.columnconfigure 236
TkGrid.rowconfigure 237
tkimg 241
tkImg パッケージ 243
TkLabel 218
TkListbox 259
TkMenubar 265
TkMessage 249
TkPhotoImage 245, 276
TkRadioButton 252
TkRoot 236
TkScale 262
TkSpinbox 264
TkText 255
TkVariable 251
tr 163

true……5
TSTP（シグナル）……78
TTY……74

■U

umask……134
uniq……162
uniq……9
unlink……126
unshift……10
upto……8
USER（環境変数）……64
USR1（シグナル）……78, 81
USR2（シグナル）……78, 81

■V

values……11

■W

wc……156
when……3
while……3
while（シェル）……181

■X

xev……222
xmodmap……221, 222

■Y

YAML……47

■あ

一時ディレクトリ……147
イベント……216
イベント駆動型プログラム……216
イベントシーケンス……220
イベントパターン……220
イベントハンドラ……219, 223
ウィジェット……217
ウィジェットのグループ化……281
ウィジェットの検索……281
円……279
折れ線……277
折れ線の頂点の形状……278
大文字小文字を同一視……20
親プロセス ID（PPID）……74

■か

画像……276
仮想イベント……227
画像オブジェクト……241
仮引数……7
環境変数……175
環境変数の取得……68
偽……5
共有ロック……54
空……5
グルーピング（正規表現）……19
グループ ID（GID）……74
グロッビング……183
弧……275
子プロセス……91
コマンドライン引数……26

■さ

最短マッチ……20
最長マッチ……20
座標の取り方……280
算術展開（シェル）……178
シーケンス……47
シェル変数……175, 182
ジオメトリマネージャ……217, 228
シグナル……78
シグナルハンドラ……78
シグナル捕捉……79
出力のバッファリング……129
シリアライズ……46
真……5
スカラ……47
スクロールバー……258
スケール……262
スピンボックス……264
スレッド……83
正規表現オプション……20
正規表現のメタキャラクタ……18
整形出力……35
制限時間付きキー入力……194
線のパターン……278

即時キー入力 193
ソフトリンク 127

■た

ダイアログ 269
楕円 279
多角形 279
タグ（TkText ウィジェット） 255
チェックボタン 251
長方形 275
ツールキット 216
データ永続化 43
テキスト 255
テキストボックス 279
デッドロック 56
同期呼び出し 91

■な

塗りつぶしの色 275

■は

ハードリンク 127
排他処理 49
排他ロック 55
バウンディングボックス 281
バッククォート 110
バッククォート（シェル） 177
ヒアドキュメント 111, 243
ビット演算子 197
非同期呼び出し 91
標準エラー出力 28
標準入力 27
ファイルテスト 136
ファイルの属性 123
ファイルポインタ 128
ファイルロック 50
フィルタコマンド 154
フォント 246
部分文字列 13
ブレース展開（シェル） 179
フレームウィジェット 238
フロースタイル（YAML） 49
プロセス 74
プロセス ID（PID） 74
ブロックスタイル（YAML） 49
フロントエンドプロセッサ 103
ボタン 250

■ま

マーク（TkText ウィジェット） 255
マッピング 47
メソッド定義 6
メニュー 265
文字クラス 18
文字属性変更 195
文字列同士の結合 13
文字列の長さ 13
モディファイアキー 221

■や

矢尻 277
矢印 277
矢印キーの利用 202
ユーザ ID（UID） 74

■ら

ラジオボタン 252
ラベル 248
リストボックス 259
リンクカウント 124
レイアウトマネージャ 217, 228
例外 141

■わ

枠の色 275
枠の線のパターン 275
枠の線の太さ 275

■ 著者プロフィール

広瀬 雄二（ひろせ・ゆうじ）

1968 年山梨県塩山市（現甲州市）生まれ。慶應義塾大学理工学研究科管理工学専攻から同インフォメーションテクノロジーセンター助手を経て、東北公益文科大学（山形県酒田市）へ。Ruby や C の書き手、ネットワークエンジニアとして育っていく卒業生は多いものの、Emacs-Lisper が未だに出ないのがすこし寂しい。かわりに実用スクリプトが作れる学生が増えてきたのは喜ばしいことである。

実用的 Ruby スクリプティング

入門から次の段階に進むためのスクリプトの書き方講座

2014 年 8 月 10 日　　初版第 1 刷発行

著　者	広瀬 雄二
発行人	石塚 勝敏
発　行	株式会社 カットシステム
	〒 169-0073　東京都新宿区百人町 4-9-7　新宿ユーエストビル 8F
	TEL（03）5348-3850　　FAX（03）5348-3851
	URL　http://www.cutt.co.jp/
	振替　00130-6-17174
印　刷	シナノ書籍印刷 株式会社

本書に関するご意見、ご質問は小社出版部宛まで文書か、sales@cutt.co.jp 宛に e-mail でお送りください。電話によるお問い合わせはご遠慮ください。また、本書の内容を超えるご質問にはお答えできませんので、あらかじめご了承ください。

■ 本書の内容の一部あるいは全部を無断で複写複製（コピー・電子入力）することは、法律で認められた場合を除き、著作者および出版者の権利の侵害になりますので、その場合はあらかじめ小社あてに許諾をお求めください。

Cover design Y.Yamaguchi　　© 2014 広瀬雄二

Printed in Japan　ISBN978-4-87783-347-3