第 10 章

状態の保存と通信

2016-06-22 Wed

概要

複数のプログラムが協調して動作する場合や，中断した計算を再開するためには，実行の状態を適当な表現に変換してやりとりする必要がある．テキスト形式のファイルの読み書きと，少し複雑なデータを json で表現することを紹介する．

10.1　事務連絡: 今後の予定

- 6 月 22 日 本日
- 6 月 29 日 これまでの補足・質問受け付け
- 7 月 6 日 これまでの補足・質問受け付け (git など便利なツールは紹介するかも)
- 7 月 13 日 休講 (学際科学科栃木実習)
- 最終課題提出締切 (提出期限は後日発表)

10.2　行単位のテキストファイルの読み書き

まず，テキストファイルを行単位で読み書きするには，`File.open` という機能を用いる．なお，単に `open` と書いても同じ意味になる．

Ruby

```ruby
# 以下のコードを write-test.rb に保存
File.open("file-test.txt", "w") {  |file|
  file.puts "hello"
  file.puts "world"
}
```

ここで中括弧でくくられたブロックにおいて，`file` は，open したファイルを指す．ブロックを抜けると，自動で close される．文中の `file.puts` は，指定したファイルを対象に `puts` を行う．通常の `puts` は，`STDOUT.puts` の略記である．`puts` 以外に，`print` なども同様.

```
$ ruby write-test.rb                                                    1
$ cat filetest.txt     # テスト                                         2
hello                                                                   3
world                                                                   4
```

なお，cat はファイルの中身を表示するコマンドである．プログラムの動作確認のために，対応するコマンドを覚えておくと良い．

Ruby

```ruby
# 以下のコードを read-test.rb に保存
File.open("file-test.txt", "r") {  |file|
  file.each { |line|
    puts line
  }
}
```

```
$ ruby read-test.rb                                                     1
hello                                                                   2
world                                                                   3
```

コード中の each は，各要素についてブロックの中を実行する構文である．each に続く中括弧のペアをブロックと呼び，ブロック冒頭の縦棒で挟んだ中に，要素を受ける変数を記述する．

Ruby

```ruby
irb(main):001:0> a = [3,1,4,1,5]
=> [3, 1, 4, 1, 5]
irb(main):002:0> a.each { |e| puts e }
3
1
4
1
5
=> [3, 1, 4, 1, 5]
```

問題　二次元配列の保存　(金子)

数の配列 x と y を，1 行に 2 つの数を空白区切りで書き出す関数 write_to_file2d(filename, x,y) を作成せよ．先頭を 0 行目として，i 行目には x[i] y[i] が出力されるとする．

テストは，y=sin(x) など適当な関数を適当な刻みで設定した配列を引数に与え，出力ファイルを gnuplot で描画する．ソースコード，元データ (あるいはデータ生成関数) とともに，画像をして添付せよ．

gnuplot の使い方は Koch 曲線で取り扱った．以下に再掲する．
ファイル名が'koch.txt' であるときに描画するには，ターミナルから以下のように行う．

```
% gnuplot
Terminal type set to 'x11'
gnuplot> plot 'koch.txt' with linespoints
```

画面で確認後に，以下のようにタイプすると画像をファイルに保存できる．

```
% gnuplot
gnuplot> set term png
gnuplot> set out 'koch.png'
gnuplot> replot
```

10.2.1 http での読み込み

Ruby では `open-uri` を読み込んでおくと，`open` に URI を解釈する機能が付与される．

Ruby

```ruby
# 以下のコードを openuri-test.rb に保存
require 'open-uri'
open("http://www.ecc.u-tokyo.ac.jp") {|f|
  f.each_line { |line|
    puts line
  }
}
```

```
$ ruby openuri-test.rb
<html lang="ja">
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <meta http-equiv="pragma" content="no-cache">
  <link rel="shortcut icon" href="/favicon.ico">
#(省略)
```

例題　HTTP 読み込み　(金子)

指定した URL の内容を，指定したファイルに保存する関数 `httpget(url, filename)` を作成せよ．

作成したプログラムを用いて，http://lecture.ecc.u-tokyo.ac.jp/~ctkaneko/pl/2016/xy.txt を保存して，gnuplot で描画してみよう．

なお，同様の操作は curl というコマンドを用いて以下のように行うことができる．また mac 以外では wget というコマンドも広く使われている．

```
$ curl -o filename 'http://...'
```

10.3 ハッシュ (連想配列)

Ruby のハッシュ (連想配列) は，データの組 (key, value) の集合を保持し，key から value を容易に検索することができる．別の言い方をすると，配列の添字は整数だが，連想配列では整数以外も添字のように使える．(以前二分木のところで簡単に取り扱った).

Ruby

```
1  irb(main):001:0> dict = {} # 連想配列の構築
2  => {}
3  irb(main):002:0> dict["hello"] = "kon-nichi-ha" # データの登録
4  => "kon-nichi-ha"
5  irb(main):003:0> dict.has_key?("hello") # key "hello"は今登録したばかり
6  => true
7  irb(main):004:0> dict.has_key?("good morning")
8  => false
9  irb(main):005:0> dict["hello"] # データの参照
10 => "kon-nichi-ha"
11 irb(main):006:0> dict["four"]
12 => nil
13 irb(main):007:0> dict["four"] = 4 # 文字列以外の型も使用可能
14 => 4
15 irb(main):008:0> dict["four"]
16 => 4
```

連想配列は，{} で構築する (配列を [] で構築したことと似ている)．キーに対応するデータの登録や参照は，配列と同様の文法を用いる．キーが登録済みかどうかは，has_key?メソッドで調べることができる．

10.3.1 配列や参照を含むハッシュ

ハッシュには，配列や別のハッシュを登録することができる．

例として，階層ディレクトリのフォルダをハッシュで表すことを考える．あるフォルダには，(フォルダでない) ファイルと，フォルダがあるので，それぞれ"files"と"folders"で表すとする

空のディレクトリは以下のように構築できる．

Ruby

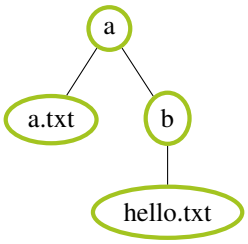
```
1 irb(main):012:0> a = {}
2 => {}
3 irb(main):013:0> a["files"] = []
4 => []
5 irb(main):014:0> a["folders"] = []
6 => []
```

```
irb(main):015:0> a
=> {"files"=>[], "folders"=>[]}
```

以下のように，ハッシュの構築時に key, value を=>記号でつないで，データを指定することもできる．

Ruby
```
irb(main):017:0> b = {"files"=>[], "folders"=>[]}
=> {"files"=>[], "folders"=>[]}
```

以下の図のような，階層関係を構築してみよう．



Ruby
```
irb(main):018:0> b["files"] << "hello.txt"
=> ["hello.txt"]
irb(main):019:0> a["files"] << "a.txt"
=> ["a.txt"]
irb(main):020:0> a["folders"] << b
=> [{"files"=>["hello.txt"], "folders"=>[]}]
irb(main):021:0> a
=> {"files"=>["a.txt"], "folders"=>[{"files"=>["hello.txt"], "folders"=>[]}]}
irb(main):022:0> b
=> {"files"=>["hello.txt"], "folders"=>[]}
```

テスト例:

Ruby
```
irb(main):023:0> a["files"][0]
=> "a.txt"
irb(main):024:0> a["folders"][0]
=> {"files"=>["hello.txt"], "folders"=>[]}
irb(main):025:0> a["folders"][0]["files"][0]
=> "hello.txt"
```

10.3.2 JSON: 階層データの文字列表現

さて，階層構造を含むようなデータは，行でデータを区切るテキストファイルで表現することが難しい．そこでウェブを中心に広く使われている JSON という表現形式を紹介する．

Ruby ではまず json ライブラリを読み込む．

Ruby
```
irb(main):006:0> require 'json'
=> true
```

これにより，配列やハッシュなどに.to_json というメソッドが追加される．

Ruby
```
1 irb(main):027:0> [3, 1, 4].to_json
2 => "[3,1,4]"
```

先ほどのディレクトリ a のような構造を持つデータも，文字列に変換される．

Ruby
```
1 irb(main):026:0> a.to_json
2 => "{\"files\":[\"a.txt\"],\"folders\":[{\"files\":[\"hello.txt\"],\"folders\":[]}]}"
```

変換された文字列は，JSON.parse という手続きにより元のデータを取り出すことができる．

Ruby
```
1 irb(main):038:0> JSON.parse("[3,1,4]")
2 => [3, 1, 4]
3 irb(main):032:0> str = a.to_json
4 => "{\"files\":[\"a.txt\"],\"folders\":[{\"files\":[\"hello.txt\"],\"folders\":[]}]}"
```

問題 **二分木の保存** (金子)

“7.1.6 内部表現の世界と使用する世界の分離” では二分木をハッシュで表現した．二分木 (たとえば Q&A で作成したもの) をファイルに保存する関数 write_tree(filename, tree) と読み込んで返す関数 read_tree(filename) を作成せよ．同じ木が復元できていることを確認せよ．

ただし，二分木の表現としては以下を使うと良い．7.1.6 とほぼ同じだが，キーとしてシンボル (たとえば :number) の代わりに文字列たとえば"number"を使うように変更してある．

Ruby
```
1  def make_node(num, left, right)
2    {"number"=>num, "left"=>left, "right"=>right}
3  end
4  def value(tree)
5    tree["number"]
6  end
7  def left(tree)
8    tree["left"]
9  end
10 def right(tree)
11   tree["right"]
12 end
```

10.4 今週の課題

2 つの問題について Ruby で取り組め．提出するソースコードが正しくインデントされ，日本語コメントが付記されていること: { 途中か完成品か，授業中に既に OK をもらっていればその教員名 } 学んだこと参考にした資料など (何も苦労がなかった場合を除き，自作したことが分かるだけの十分なエピソードを記述すること – 授業中に教員の OK をもらった場合は簡潔で良い)

付録 A

最終課題資料

最終課題は，これまでの演習で練習した技術を組み合わせて多少規模の大きなプログラムを作ることと，既存のプログラムとあるいは他者のプログラムと組み合わせて動かすことを主眼にする．

以下のいずれかを，夏学期試験最終日までに提出のこと:

1. Tron という同時着手ゲームの思考部分を作る．対戦フレームワークは指定する (既存の) ソースコードを流用し，それに合わせて作ること．
2. 対戦フレームワーク部分を自分で作成したコードで置き換え，他の人が作った思考部分同士を対戦させられるようにする．マップや時間制限の指定，GUI での表現や成績管理機能などがあるとなお良い．
3. 3 人程度でチームを組んで，独自のゲーム (同時着手ゲームもしくは不完全情報ゲームが望ましい) の対戦フレームワークとプレイヤを分担して作成する．

この資料では，選択肢 1 のみを説明する．選択肢 2 および 3 は，適宜資料を用意するので担当教員に相談のこと．

A.1 Tron の思考部作成の手引

目標

- 必須条件 「自分から壁にぶつからない」「他の人と戦略が少しはことなる」
- 努力目標 なるべく強く

環境設定と対戦

1. 教材から tron-ruby-ecc.zip をダウンロードして、適当な場所に展開する
2. ターミナルから以下を実行

```
$ cd ruby-ecc                                                    1
$ java -jar engine/Tron.jar maps/empty-room.txt \                2
 "java␣-jar␣example_bots/RandomBot.jar" "ruby␣MyTronBot.rb"      3
```

- 2 行目の \ は継続行の印なので，3 行目を続けてタイプする．
- `maps/empty-room.txt` 部屋の形状

- "java -jar example_bots/RandomBot.jar" 第一プレイヤ
- "ruby MyTronBot.rb" 第二プレイヤ

(maps や example_bots 以下は適当に指定可)

作り方: MyTronBot.rb を改造する

A.2 サンプルプログラムを読む

サンプルプログラムである MyTronBot.rb で，次の動きを決める関数 makemove は以下のようになっている．まずはこれを理解しよう．

Ruby

```ruby
def makemove(map)
  # プレイヤの現在地を取得
  x, y = map.my_position
  # 上下左右のうち行ける方向を配列に格納
  valid_moves = []
  valid_moves << :NORTH if not map.wall?(x, y-1)
  valid_moves << :SOUTH if not map.wall?(x, y+1)
  valid_moves << :WEST  if not map.wall?(x-1, y)
  valid_moves << :EAST  if not map.wall?(x+1, y)

  if(valid_moves.size == 0)
      # どこにも動けない?
      map.make_move( :NORTH )
  else
      # ランダムに決めよう
      move = valid_moves[rand(valid_moves.size)]
      # uncomment to show move
      # puts move
      map.make_move( move )
  end
end
```

これまでに紹介していない Ruby の文法は以下のとおりである．

■多重代入　代入演算子=の左右にカンマで区切られた式が同数あるときに，それぞれ代入が行われる．

```
irb(main):001:0> a, b = 3,5
=> [3, 5]
irb(main):002:0> a
=> 3
irb(main):003:0> b
=> 5
```

変数の値を交換したい場合に便利である

```
irb(main):008:0> a, b = b, a
=> [5, 3]
irb(main):009:0> a
=> 5
irb(main):010:0> b
=> 3
```

■短い if 文　これまでに紹介した `if` は次のように用いた．

```
if 条件
  条件が成り立ったら実行する文
end
```

等価な表現として以下も使用可能である．

```
条件が成り立ったら実行する文 if 条件
```

```
irb(main):012:0> print "hello\n" if 5 % 2 == 1
hello
=> nil
irb(main):013:0> print "hello\n" if 5 % 2 == 0
=> nil
```

■シンボル　`:NORTH` などのコロン`:`に文字を続けたものは，シンボルである．ここでは変更不可能な文字列と理解することにして深入りしない．

■その他　条件の否定を`!`でなく `not` と書くこともできる．また関数やメソッドの名前には英数字だけでなく`?`,`!`などの記号も利用可能である．判定するだけでデータの変更を行わない関数には`?`を，破壊的変更を行う関数には`!`を使う慣習がある．

```
irb(main):014:0> print "hello\n" if not 5 % 2 == 0
hello
=> nil
irb(main):015:0> array = [3,2,1]
=> [3, 2, 1]
irb(main):016:0> array.sort # 新しい配列を返す
=> [1, 2, 3]
irb(main):017:0> array
=> [3, 2, 1]
irb(main):018:0> array.sort! # array 自身を変更
=> [1, 2, 3]
irb(main):019:0> array
=> [1, 2, 3]
```