特集 マルチコアを活かすお手軽並列プログラミング

5 dRuby と Rinda ― Ruby における並列計算

関　将俊（druby.org）

dRuby

本稿は dRuby と Rinda について報告するものである．dRuby は Ruby に分散オブジェクト環境を提供し，Rinda は dRuby を用いた協調機構である．

dRuby を紹介する前に，Ruby について述べる．Ruby はまつもとゆきひろ氏によって開発されたオブジェクト指向スクリプト言語である．これまで先鋭的なプログラマから強く支持されていたが，Ruby on Rails の登場によってビジネスとして利用する人々からも注目されるようになった．Ruby は動的なオブジェクト指向言語である．すべてがオブジェクトで，変数に型はなく，メソッドの結合は実行時に行われる．また，豊富なリフレクション機能を持ち，メタプログラミングも可能である．

そして dRuby はこの Ruby の分散オブジェクト環境である．dRuby は Ruby のメソッド呼び出しをネットワーク上に拡張し，別のプロセス／マシンのオブジェクトのメソッドを利用可能とする．さらにもう 1 つの重要なライブラリは Rinda である．Rinda は並列プログラムを協調させるためのモデル Linda を dRuby を用いて実装したものである．本稿は，dRuby の概念から設計ポリシー，実世界の応用例まで紹介するものである．

dRuby は Ruby の RMI（Remote Method Invocation）を提供するライブラリの 1 つである．筆者が dRuby の開発で目指したのは既存の分散システムの Ruby 化ではない．Ruby のメソッド呼び出しをネットワークを通じて他のプロセス／マシンへ拡張することである．

dRuby には次のような特徴がある．

- Ruby に限定
- IDL（Interface Definition Language）などのインタフェース定義が不要
- セットアップ，習得が容易
- オブジェクト交換方式（値渡しと参照渡し）の自動選択機構
- プロセスはサーバとしてもクライアントとしても振る舞う

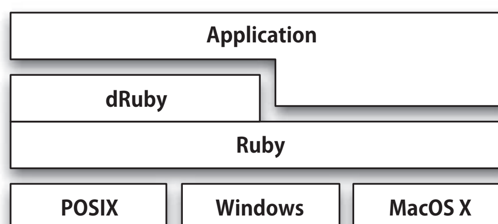

図 -1　dRuby のアプリケーションの構成

dRuby はすべて Ruby スクリプトで記述されている．Ruby には優れたスレッド，ソケット，マーシャリングのクラスライブラリが備わっているため，最初のバージョンの dRuby はわずか 200 行で実装することができた．この事実は Ruby のクラスライブラリの強力さとセンスの良さを示すものである．

dRuby は Ruby の配布パッケージに標準で含まれており，Ruby がインストールされているならすぐに dRuby を利用可能である．dRuby は Ruby が動作すれば CPU や OS を問わず利用できる（図 -1）．すなわち，プラットフォームが異なっていたとしても，Ruby が動作すれば，お互いのオブジェクトのメソッドを利用し，オブジェクトを交換できるということである．

❒ dRuby を使う

はじめにスレッドの同期の例を通じて dRuby の簡単な使い方を示す．並行プログラミングのよく知られた問題に生産者消費者問題がある．有限のバッファに対

しデータを書き込む生産者と，データを読み込む消費者の2つのプロセスを協調させる問題である．以下は長さの上限が設定できる待ち行列，SizedQueue クラスを使って生産者消費者問題を解く（dRuby を使わずにスレッドを使った）スクリプトである．まず SizedQueue を利用するために 'thread' ライブラリをロードする（1行目）．次に producer メソッドと consumer メソッドの定義がある（3～15行目）．どちらもランダムに休眠しながら SizedQueue への操作を100回繰り返す．SizedQueue は長さの上限が設定されているため，producer はデータを投入しすぎることはない．次に producer と consumer の間を取り持つ SizedQueue オブジェクトを生成する（17行目）．並行して処理が動くように producer は新しいスレッドで起動する（18～20行目）．そして最後に consumer メソッドを呼び出す．

mono.rb
```
 1 require 'thread'
 2
 3 def producer(queue)
 4   100.times do |n|
 5     sleep(rand)
 6     queue.push(n)
 7   end
 8 end
 9
10 def consumer(queue)
11   100.times do |n|
12     sleep(rand)
13     puts queue.pop
14   end
15 end
16
17 queue = SizedQueue.new(10)
18 Thread.new do
19   producer(queue)

20 end
21 cosumer(queue)
```

Ruby のスレッドはユーザレベルのスレッドであり，同時に1つのスレッドしか実行されない．現在のところ[☆1]Ruby で並行に処理をさせたい場合にはプロセスを分けるのが一般的である．dRuby はそのような複数の Ruby のプロセスを連携させるのに向いている．

先のスクリプトを dRuby を用いて機能ごと（producer，consumer，SizedQueue）に分割してみよう．はじめに SizedQueue サーバ（queue.rb）を説明する．このプロセスは producer，consumer から利用される SizedQueue のサーバである．

queue.rb
```
1 require 'thread'
2 require 'drb/drb'
3
4 queue = SizedQueue.new(10)
5 DRb.start_service(
6   'druby://localhost:49999',
7   queue)
8 sleep
```

dRuby を利用するアプリケーションははじめに 'drb/drb' をロードする（2行目）．SizedQueue オブジェクトを生成し，DRb.start_service によって dRuby のサービスを開始する（5行目）．DRb.start_service は，他のプロセスに公開したいオブジェクトと，その URI を与えることができる．この呼び出しは，SizedQueue オブジェクトを 'druby://localhost:49999' という URI で公開することを示している．

dRuby ではどのオブジェクトも他のプロセスに公開することができるが，他のプロセスが目的のオブジェクトに到達するためにはこういった明示的な名前を持つオブジェクトが必要となる．dRuby によって作られるシステムでは，どこかに必ず，システムの入り口を示すオブジェクトが存在する．その入り口となるオブジェクトを front オブジェクトと呼ぶ（**図-2**）．

最後にこのプロセスが終了してしまわないように sleep で停止する（8行目）．dRuby のサービスはバックグラウンドのスレッドで行われるため，sleep 中も問題なく RMI を受け付けることができる．もし sleep を忘れるとプロセスが直ちに終了してしまうので注意が必要である．また，RMI は独立したスレッドで処理されるため，同時に複数の RMI を扱うことができる．これは，pop で中断している最中に push を処理することができることを意味する．もし一度に1つの RMI しか処理できないのであれば，pop で中断したプロセスは永遠に再開できないため，SizedQueue サーバは生産者消費者問題に使えないことになってしまう．

producer.rb は SizedQueue サーバの待ち行列にデータを投入するプロセスである．

producer.rb
```
1 require 'drb/drb'
2
3 def producer(queue)
4   100.times do |n|
5     sleep(rand)
6     queue.push(n)
7   end
```

☆1 1つのプロセスで複数のインタプリタが同時に実行されるマルチ VM が開発中である．

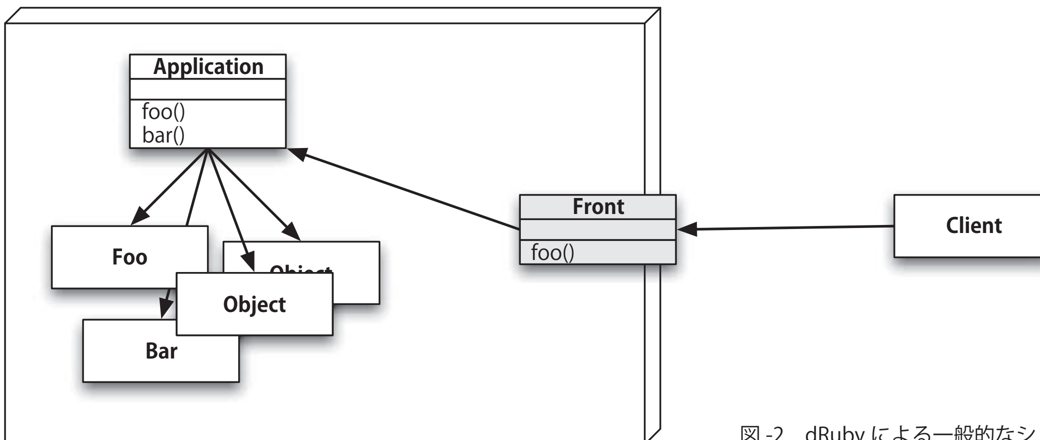

図 -2　dRuby による一般的なシステムと front オブジェクト

```
 8 end
 9
10 DRb.start_service
11 queue = DRbObject.new_with_uri(
12   'druby://localhost:49999')
13 producer(queue)
```

はじめに DRb.start_service で dRuby を初期化する（10 行目）．SizedQueue サーバに対してクライアント的なプロセスではあるが，dRuby に参加するプロセスはみな DRb.start_service を行う．これは暗黙的なオブジェクトの公開が発生することがあるためである．暗黙的なオブジェクトの公開については後述する．

次に SizedQueue サーバが公開するオブジェクトへの参照を明示的に作成する（11 行目）．URI を与えて DRbObject を生成することでその URI が示す dRuby サーバの front オブジェクト，この例では SizedQueue オブジェクトを利用できるようになる．参照を生成する際に，目的のオブジェクトがどのようなクラスであるか，どのようなインタフェースを持つかを与えていない点に注目してほしい．dRuby ではどのようなオブジェクトであるかを知ることなく参照をつくることができる．

consumer.rb も producer.rb と同様，はじめに dRuby のサービスを起動し（10 行目），SizedQueue への参照を生成し（11 行目）．最後に consumer メソッドを呼ぶ（13 行目）．

consumer.rb

```
 1 require 'drb/drb'
 2
 3 def consumer(queue)
 4   100.times do |n|
 5     sleep(rand)
 6     puts queue.pop
 7   end
 8 end
 9
10 DRb.start_service
11 queue = DRbObject.new_with_uri(
12   'druby://localhost:49999')
13 consumer(queue)
```

実行するための特別なセットアップは何もない．3 つのターミナルを用意して queue.rb，consumer.rb，producer.rb の順に起動するだけである．このサンプルコードはスレッド間の同期メカニズムである SizedQueue が，dRuby を介して他のプロセスの同期メカニズムとしても使用することができることと，普通の Ruby のアプリケーションを dRuby によって簡単に分散オブジェクト化できることを示している．dRuby は Ruby でアプリケーションを書くように，簡単に分散システムを記述することができ，分散システムのアーキテクチャのスケッチや学習に最適である．さらにそのまま実運用可能なシステムとすることも可能である．

Rinda

次に dRuby を元にした Linda[1), 2)] の実装，Rinda を紹介する．Linda は並列プログラムを協調させるためのモデルで，シンプルでありながら並列プログラムの開発において出会うさまざまな状況に対応できる魅力的なモデルである．

Linda のモデルはメモリモデルである．タプルというデータと，タプルスペースというタプルのバッグ（重複のある順序のない集合）で構成される．Linda はあくまでモデル，つまり問題についての考え方である．Linda の実装の仕方はさまざまで，OS として実装されることや言語のプリプロセッサとして実装されることもある．また，言語に合わせて機能を置き換えることもある．C-Linda や JavaSpaces，Rinda は代表的な実装の例で

ある．

Rubyのタプルスペースの実装はRindaである．Rindaは Lindaのタプルとタプルスペースのモデルを Rubyのクラスライブラリとして実装したものだ．Rindaでは，Lindaの基本操作のうち，evalを除いたout, in, inp, rd, rdpが利用できる．Rindaにはevalは用意されないが，おそらくRubyスレッドで代用できるであろう．

Rindaの最近のバージョンでは，基本操作のメソッド名がJavaSpaces風に変更された．

write（または out）　タプルをタプルスペースに追加する．

take（または in）　マッチしたタプルをタペルスペースから削除し，削除したタプルを返す．マッチするタプルがなければ，マッチするタプルがwriteされるまでブロックする．

read（または rd）　マッチしたタプルのコピーを返す．マッチするタプルがなければ，マッチするタプルがwriteされるまでブロックする．

takeとreadはタイムアウト時間を設定できる．タイムアウト時間に0を設定することで，inp, rdpと同様に動作する．

Rindaを使った疑似コードでLindaのモデルを説明しよう．まず，writeはタプルをタプルスペースに追加する操作である．タプルはオブジェクトの並びであり，RubyのArrayを用いて表現する．なお「:」で始まる値はRubyのSymbolのリテラルである．

```
ts.write([:chopstick, 1])
ts.write([:matrix, 1.6, 3.14])
```

takeはパターンにマッチするタプルをタプルスペースから取り除き，そのタプルを返す操作である．パターンもタプル同様にオブジェクトの並びで，Arrayを用いて表現する．パターンの要素のうち，nilである要素に関してはワイルドカードと見なしあらゆるオブジェクトと適合する．それ意外の要素については「===（case equals）」による比較を行う．case equalsはRubyの制御構文caseで利用される比較方法で，通常の==による比較のほか，クラスとインスタンスの関係や，範囲オブジェクト，正規表現による文字列の走査などの比較も行う．簡単な問合せ言語のような使い方も可能である．

もしtakeのパターンにマッチするタプルがタプルスペースに存在しない場合，takeを行ったプロセスの実行は中断される．他のプロセスがそのパターンにマッチするタプルを追加したとき処理が再開される．マッチするタプルがタプルスペースの中に複数ある場合，その中の1つが適当に選択される．

```
ts.take([:chopstick, nil])
  # => [:chopstick, 1]
ts.take([:matrix, Numeric, Numeric])
  # => [:matrix, 1.6, 3.14]
```

readはtakeと同じようにパターンにマッチするタプルを返す操作であるが，そのタプルをタプルスペースから取り除かない．もしマッチするタプルがタプルスペースに存在しない場合，そのようなタプルが追加されるまで処理を中断する．

N-Queens 問題

dRubyとRindaの応用例として，本特集の共通課題であるN-Queens問題をデータ並列で記述する．端の2行の組合せを与え，それぞれについて解を数え，最後にその合計を求める戦略である．なお，端の2行の組合せにおいて解がないものを除かないとする．はじめに，逐次的な数え方をnq.rbに示す．nqメソッドで数えることもできるが，今回の課題のために2行の組合せを起点として数えるnq2メソッドを用意した．なお，スクリプトの最後の分岐はクラスライブラリを単体で実行した場合に処理を行うイディオムである．nq.rbを直接実行した場合にだけこの条件が真となり，サンプルスクリプトを実行する．

nq.rb

```
 1 module NQueen
 2   module_function
 3   def concat(board, row)
 4     board.each_with_index do |v, col|
 5       chk = (v - row).abs
 6       return nil if chk == 0
 7       return nil if chk == board.size - col
 8     end
 9     board + [row]
10   end
11
12   def nq(size, board=[])
13     found = 0
14     size.times do |row|
15       fwd = concat(board, row)
16       next unless fwd
17       return 1 if fwd.size == size
18       found += nq(size, fwd)
19     end
20     found
21   end
22
23   def nq2(size, r1, r2)
24     board = concat([r1], r2)
25     return 0 unless board
```

```
26     nq(size, board)
27   end
28 end
29
30 if __FILE__ == $0
31   size = (ARGV.shift || '5').to_i
32   puts NQueen.nq(size)
33 end
```

次に Rinda を用いた並列版を示す．この並列版はタプルスペース，2 種類のタプル，2 種類のプロセスで構成する．タプルは処理を依頼するリクエストのタプルと，その結果を示すレスポンスのタプルである．リクエストのタプルは識別子（:nq）と盤のサイズ，端の 2 行の列を示す整数からから構成される．レスポンスのタプルは識別子（:nq_ans）と盤のサイズ，端の 2 行の列を示す整数，発見した解の数から構成される．

[:nq, 盤のサイズ , 行 1, 行 2]

[:nq-qns, 盤のサイズ , 行 1, 行 2, 解の数]

2 種類のプロセスはリクエストのタプルを受け取り，解の数をレスポンスのタプルに入れて返却するエンジン部（nqe.rb）と，端の 2 行に対応するデータを投入し，結果を合計するコントロール部（nqr.rb）である．

エンジン部を見てみよう．まず，dRuby を用いてタプルスペースサーバへの参照オブジェクトを生成する（5 ～ 7 行目）．次にタプルスペースからリクエストのタプル（[:nq, Integer, Integer, Integer]）を 1 つ取り出し，解の数を数え，レスポンスのタプル（[:nq_ans, size, r1, r2, found]）を投入する処理を繰り返す．

nqe.rb
```
 1 require 'rinda/rinda'
 2 require 'nq'
 3
 4 DRb.start_service
 5 ro = DRbObject.new_with_uri(
 6   'druby://localhost:49996')
 7 ts = Rinda::TupleSpaceProxy.new(ro)
 8
 9 while true
10   sym, size, r1, r2 =
11     ts.take([:nq,
12               Integer, Integer, Integer])
13   found = NQueen.nq2(size, r1, r2)
14   ts.write([:nq_ans, size, r1, r2, found])
15 end
```

コントロール部を示す．タプルスペースの参照を作るのはエンジン部と同様である（4 ～ 6 行目）．はじめに端の 2 行のすべての組合せについてリクエストのタプルを投入する（9 ～ 13 行目）．そして，すべての組合せについてレスポンスのタプルを回収し（16 ～ 23 行目），解の数を合計する．

エンジン部はデータを 1 つずつ取り出し，結果を返却するのに対し，コントロール部はすべてのデータを投入したのちに，すべてのデータを回収し解の数を印字する．

nqr.rb
```
 1 require 'rinda/rinda'
 2
 3 DRb.start_service
 4 ro = DRbObject.new_with_uri(
 5   'druby://localhost:49996')
 6 ts = Rinda::TupleSpaceProxy.new(ro)
 7
 8 size = (ARGV.shift || '5').to_i
 9 size.times do |r1|
10   size.times do |r2|
11     ts.write([:nq, size, r1, r2])
12   end
13 end
14
15 found = 0
16 size.times do |r1|
17   size.times do |r2|
18     tuple = ts.take([:nq_ans,
19                      size, r1, r2,
20                      nil])
21     found += tuple[4]
22   end
23 end
24 puts found
```

これら 2 種類のプロセスのほかにタプルスペースサーバ（place.rb）が必要となる．

place.rb
```
 1 require 'rinda/tuplespace'
 2
 3 ts = Rinda::TupleSpace.new
 4 DRb.start_service(
 5   'druby://localhost:49996',
 6   ts)
 7 DRb.thread.join
```

place.rb を最初に起動したあとは，任意の順序で nqe.rb と ner.rb を起動すればよい．エンジン部は必要に応じて複数起動できる．コントロール部とエンジン部はタプルスペースを介して協調するので，エンジン部がいくつ起動されているか気にする必要はない．Core 数に応じてエンジン部のプロセスを増やすと処理時間の短縮が期待できる．エンジン部のプロセス数によってどのように処理時間が変化するか，QuadCore を 2 つ搭載する合計 8Core の計算機で実験した．エンジン部のプロセス数に比例して処理時間が短縮されるが，Core 数以

プロセス数	1	2	3	4	5	6	7	8	9	10	11	12
実行時間 (sec)	69.06	35.00	23.01	17.43	14.20	11.79	10.38	9.37	9.28	9.36	9.53	9.62

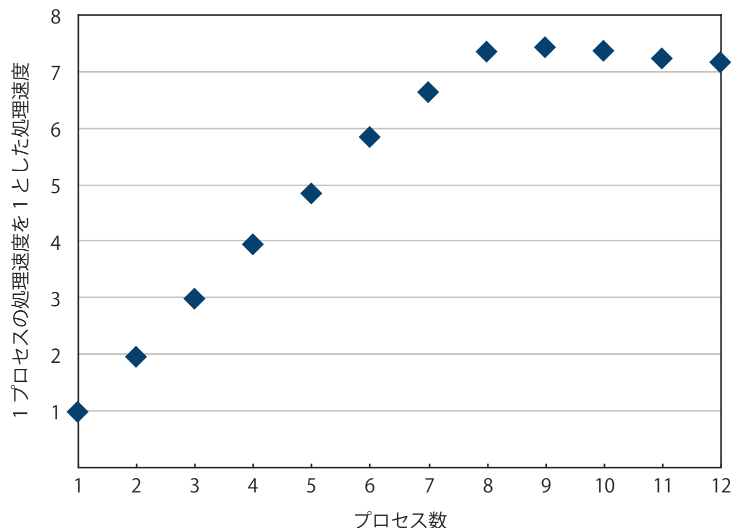

図-3　プロセス数と実行時間

上に増やしても処理時間は短縮されないことが分かる（図-3）．なお，問題の盤面の大きさは12である．

dRubyの詳細

この章ではdRubyの設計指針と実装の一部を説明したい．Rubyの経験があればdRubyをすぐに利用することができるのは，dRubyの内部ではさまざまな工夫が行われているからである．これらの仕組みを知っておくことは分散オブジェクトのシステムを設計するために大変重要である．特にオブジェクトの交換の規則と暗黙的なオブジェクトの公開の関係については読者に伝えておかなくてはならない．

❒ Rubyとの互換性

dRubyはRubyスクリプトとの互換性に特に注意を払っている．dRubyはRubyの感触を残したまま，分散オブジェクトシステムを追加した．dRubyはRubyプログラマにとって，親しみやすく，シームレスな存在である．その反面，一般的な分散オブジェクトシステムのプログラマからは奇妙に感じられるだろう．

Rubyの変数には型がなく，継承関係による代入の制約がない．静的な型を持つJavaなどのように実行前にオブジェクトの正当性を検証せず，メソッドの検索はすべて実行時（メソッド呼び出し時）に行われる．この性質はRubyの大きな特徴の1つである．dRubyにおいても同様である．dRubyのプロキシオブジェクト（dRubyでは参照またはDRbObjectと呼ぶ）にも型はなく，メソッドの検索はすべて実行時に行われる．公開されたメソッドの一覧や，継承情報などを事前に互いに知っておく必要がないのである．このため，IDLなどによるインタフェースの定義を必要としない．

dRubyはRubyのメソッド呼び出しをネットワーク上に拡張する以外には，できるだけRubyの仕様を変更しないように注意している．この指針によって，Rubyの特徴を存分に享受できるのである．たとえば，イテレータと呼ばれるブロック付きメソッドも利用できるし，例外もそのまま使用できる．MutexやQueueなどのスレッド間同期メカニズムも，そのままプロセス間の同期に使うことができる．

❒ オブジェクトの交換

dRubyはRubyの振舞いを変更しないように工夫しているが，Rubyに存在しない概念を導入しなくてはならない場面がある．そういった場合でもできるだけ自然に感じられるように配慮した．たとえば，オブジェクトの転送方式である．メソッド呼び出しにおける引数や，戻り値，例外などではオブジェクトが転送される．引数はクライアントからサーバへ，戻り値や例外はサーバからクライアントへのオブジェクトの転送である．本稿ではこれらの転送をまとめてオブジェクトの交換と呼ぶことにする．

Rubyの変数に代入できるのは参照だけである．オブジェクトのクローンが代入されることはない．しかし，dRubyにおいてはクローンの代入が発生することがありRubyの振舞いと異なってしまう．おそらく，分散オブジェクトの世界では「値を渡す」「参照を渡す」といっ

た違いが存在してしまうことは避けられないだろう．この違いは dRuby も同様に避けられない．Ruby と同様に永遠に参照を交換し合う計算モデルも考えられるが，現実的なアプリケーションではいつか「値」が必要になる．dRuby ではできるだけプログラマに転送方式の違いを意識させず，かつ現実的な効率のよさを考慮した仕組みを用意した．dRuby ではプログラマが明示的に参照渡しか値渡しかを指示するのではなく，システムが自動的に判断して参照渡しか値渡しかを選択するのである．dRuby はオブジェクトがシリアライズ可能かどうかをその判断の基準とした．

常にこの判断が正しいとは言えないが，多くの場合にうまく動作する．この規則について少し補足したい．まず，シリアライズできないオブジェクトはコピーを渡すことができないため値渡し不能であり，参照渡しを選択するしかない．問題となるのは，参照渡しにしたほうが都合がよいものが，値渡しとなるケースである．このために，dRuby はシリアライズ可能なオブジェクトを参照渡しであると表明する機能が準備されている．

この自動的なオブジェクトの転送方式の選択機能によって，dRuby ではオブジェクトの転送方式に関してプログラマが書かなくてはならないスクリプトをきわめて少ないものとしている．

ブロック付きメソッドの例を使い，自動的にオブジェクトの転送方式が選択される様子を観察しよう．Ruby ではメソッドへの引数としてオブジェクトだけでなく無名のメソッド(ブロック)を与えることができる．ブロックを引数として受け付けるメソッドをブロック付きメソッドと呼ぶ．ブロック付きメソッドは集合の要素に対する操作で多く使用されることから，イテレータと呼ばれることもある．以下はブロック付きメソッドの例である．指定された大きなファイルから 4096 バイトずつ順にブロックへ渡し(6 ～ 13 行目)処理するスクリプトだ．

jukebox.rb

```
 1 class JukeBox
 2   def initialize(dir)
 3     @dir = dir
 4   end
 5
 6   def stream(name)
 7     fname = File.join(@dir, name)
 8     File.open(fname) do |f|
 9       while buf = f.read(4096)
10         yield(buf)
11       end
12     end
13   end
14 end
15
16 if __FILE__ == $0
17   jukebox = JukeBox.new("./your/songs")
18   jukebox.stream(ARGV.shift) do |buf|
19     $stdout.write(buf)
20   end
21 end
```

次にこの例題を dRuby を用いて分割する．まず jukebox サーバを示す．JukeBox オブジェクトを生成し，URI と関連づける．

server.rb

```
1 require 'jukebox'
2 require 'drb/drb'
3
4 jukebox = JukeBox.new("./your/songs")
5 DRb.start_service(
6   'druby://localhost:45000',
7   jukebox)
8 sleep
```

続いてクライアントを示す．

client.rb

```
1 require 'drb/drb'
2
3 DRb.start_service
4 jukebox = DRbObject.new_with_uri(
5   'druby://localhost:45000')
6 jukebox.stream(ARGV.shift) do |buf|
7   $stdout.write(buf)
8 end
```

クライアントはリモートのオブジェクト jukebox に対して，ブロック付きメソッドを呼び出しているが，その引数は目的のファイル名(ARGV.shift)と do ... end のブロックの 2 つである．ファイル名は String であるのでシリアライズされ値渡しとなる．しかし，ブロックはシリアライズができないため，自動的に参照渡しが選択される．つまりブロックは jukebox サーバに暗黙的に公開されることとなる．jukebox サーバは jukebox.rb の yield により，この参照を呼び返す．このとき，jukebox クライアントはサーバとして振る舞い，jukebox サーバからのメソッド呼び出しを処理する．

IDL などのインタフェース定義やオブジェクトの転送方法の宣言などのほかにも，一般的な分散オブジェクトシステムと異なっている点は少なくない．dRuby は「Ruby のような分散オブジェクトシステム」となることを目指しているからである．

❒ サポートしていない事柄

最後にdRubyがサポートしていない事柄も紹介しなくてはならないだろう．ガーベージコレクション（以下GC）とセキュリティだ．dRubyでは分散GCを実装していない．安価で現実的な解決案がまだ見つからないためだ．現在のところ，GCされないように気をつけるのはアプリケーションの責任である．ping方式によってGCから保護する方法はオプションとして用意されるが，循環参照によって消えないオブジェクトが生まれる可能性がある．Rubyインタプリタに手を加えて，この問題を解決する研究も行われている．dRubyはセキュリティに関してもなにもしない．Rubyにならい，Rubyと同様なメソッドの可視性の制御が行われる程度であり，悪意のある攻撃に対して無力である．なお，通信経路としてSSLを選ぶことは可能である．

本章ではdRubyの設計指針について説明した．dRubyはRubyのメソッド呼び出しをそのまま拡張するものであり，一般的なRMIのRuby風インタフェースではない．だからこそdRubyとXML-RPC，SOAP，CORBAなどと競合するものではなく，むしろ共存するものであると考える．実際，外部のネットワークとのインタフェースとしてhttpを使い，そのバックエンドにdRubyで構成された内部システムを持つものも多く存在する．

アプリケーション

dRubyを利用したアプリケーションは多い．Ruby on Railsではデバッガの実装，Webアプリケーションの非同期処理などで数多く利用されていることはよく知られている．この章ではdRubyとRindaを用いたアプリケーションの例を紹介する．

❒ 大規模Webアプリケーションのバックエンド

大規模Webアプリケーションのバックエンドの例として，RubyKaigi 2006において，舘野氏によって報告された“はてなスクリーンショットサービス”を紹介する．このサービスは登録されたURLのスクリーンショットを，blogなど他のはてなのサービスにサムネイルなどを表示するものである．Webフロントエンドは Linux上に構築されるが，スクリーンショットの撮影はWindowsのIEコンポーネントを用いて実現する．Windowsで撮影するのは，Windows環境の方が高速にスクリーンショットを撮影できるためである．スクリーンショットはフロントエンドとは非同期にバッチ処理として実行される．Windows上で動作するプロセスは，Linux上のプロセスからdRubyを介してURLと返却方法をオブジェクトとして受け取り，スクリーンショットを撮影する．Windowsマシンは2台用意され，並列に処理が行われる．スピードが必要になったらWindowsマシンを追加することで対応できる構成である．

2006年のRubyKaigiの発表当時のデータによると，このシステムの扱う問題の規模は，2台の並行処理により120screenshot(ss)/min，170,000ss/dayとのことである．

❒ Rindaの応用

実世界でのRindaの応用例としてBuzztterを紹介する．Buzztterは，短文に特化したSNS（Social Networking Service）であるTwitterの文章を解釈するWebサービスである．BuzztterはTwitterに投稿される文章を集めて解釈し，普段よりも多く言及されている言葉を見つけ出すことで，Twitter参加者の動向を推測する．Buzztterはいくつかのサブシステムで構成される．Rindaを利用しているのは，Twitter API（HTTP）を用いて文章を集めてくる分散クローラサブシステムである．このサブシステムはTwitterから情報をフェッチする複数のfetcherと，永続化を行うimporterによって構成される．fetcherとimporterの間を取り持つのがRindaとdRubyである．

❒ 実世界へ

dRubyの概念から設計ポリシーについて説明し，dRubyをベースに開発されたタプルスペースの実装，Rindaについて述べた．dRubyもRindaも，Rubyプログラマにとって馴染みやすいシンプルなシステムとして設計されており，分散システムのスケッチに最適である．しかしながら，多くの実世界の応用例は，dRubyおよびRindaがスケッチ専用の「おもちゃ（toy）」ではなく，実世界のアプリケーションを構築するインフラとして利用できることを示している．

参考文献

1）Carriero, N. and Gelernter, D. : Linda in Context, Commun. ACM, Vol.32, No.4, pp.444-458 (1989).
2）関　将俊 : dRubyによる分散・Webプログラミング，オーム社 (2005).

（平成20年9月15日受付）

関　　将俊　m_seki@mva.biglobe.ne.jp

プログラマ．1992年東芝医用システムエンジニアリング（株）入社．druby.org．主な著作「dRubyによる分散・Webプログラミング」．