オブジェクト指向技術'11 ＃4 — 分散システムとOO

久野　靖*

2011.5.12

1　SableCCコンパイラフレームワーク (つづき)

1.1　構文解析ふたたび

2週間空いたので構文解析のところから復習するのですが、同じ例ではつまらないので、日本語を使ったヘンな言語を定義してみます。

```
Package sem3;
Helpers
  digit = ['0'..'9'];
  lcase = ['a'..'z'];
  ucase = ['A'..'Z'];
  letter = lcase | ucase;
Tokens
  woireru = 'を入れる';
  niyomikomu = 'に読み込む';      ←「に」が先だとまずいので注意
  ni = 'に';                        (先に書かれているものが優先なので)
  wouchidasu = 'を打ち出す';
  naraba = 'ならば';
  wojikkou = 'を実行';
  noaida = 'の間';
  gt = '>';
  lt = '<';
  add = '+';
  sub = '-';
  number = digit+;
  ident = letter (letter|digit)*;
  blank = (' ' | 10 | 13 )+;

Ignored Tokens
  blank;

Productions
  prog = {stlist} stlist
       ;
  stlist = {empty}
```

*経営システム科学専攻

```
      | {stat} stlist stat
      ;
stat = {assign} ident ni expr woireru
      | {read}   ident niyomikomu
      | {print}  expr wouchidasu
      | {if}     expr naraba stlist wojikkou
      | {while}  expr noaida stlist wojikkou
      ;
expr = {term} term
      | {gt} [left]:term gt [right]:term
      | {lt} [left]:term lt [right]:term
      ;
term = {fact} fact
      | {add}  term add fact
      | {sub}  term sub fact
      ;
fact = {ident} ident
      | {number} number
      ;
```

この言語のプログラム例を示します。トークンの間は空けてもいい (というか空けた方が読みやすい) のですが、日本語ぽくわざとくっつけて書いてみました。

```
n に読み込む
x に 1 を入れる
n>0 の間 x に x+x を入れる n に n-1 を入れるを実行
x を打ち出す
```

演習 1 次のような計算をするプログラムをこの言語で書け。

a. 2 つの数を読み込んで合計を打ち出す。
b. 2 つの数を読み込んで大きい順に打ち出す (2 数は等しくないものとしてよい)
c. 入力した値 n を超えないフィボナッチ数を順番に打ち出す。

1.2 構文木のたどり

いよいよ、構文木をたどって動作を行う部分を見てみましょう。既に述べてきたように、SableCC では構文木はパーサによって自動的に作られ、それをたどるのには拡張された Visitor パターンが使われています。Visitor の土台となるクラスとして `DepthFirstAdapter` というクラスが生成されていて、ここには文法に現れるすべてのノードの visit メソッドが予め「何もしない」形で用意されているので、このクラスを継承して必要なところだけをオーバライドしていくことで必要な処理を記述します。

オーバライドするためには、メソッド名が分かっている必要がありますね。SableCC では、構文規則に対応してメソッド名が次のように決められます。まず構文規則が次のものだとします。

```
xxx : {yyy} aa bb cc
    | {zzz} [left]:aa bb [right]:aa
    ;
```

まずノードクラスについて説明しましょう。1 つのノードは 1 つの規則に対応しているので、上の場合は 2 つのノードオブジェクトが定義されています。それらのクラス名はそれぞれ、「AYyyXxx」と「AZzzXxx」になります (先頭が A、次が {} 内に書かれた規則の名前を Capitalize したもの、次が左辺の記号名を Capitalize したもの)。

そして、これらのノードクラスはそれぞれ、右辺の各要素を取り出すメソッドとして getAa()、getBb()、getCc() の 3 つ、および getLeft()、getBb()、getRight() の 3 つを持ちます (このため、同じ名前の記号に対しては区別のための別の名前を指定する必要があったわけです)。これらが返すのはそれぞれのノードオブジェクトですが、そのノードが端記号の場合はその端記号に対応していた文字列が getText() によって取得できます。

いよいよ Visitor のためのメソッドですが、これは DepthFirstAdapter において各ノードごとに 3 つのメソッドが用意されています。たとえば上の例で 1 番目のノードでは次のようになります。

```
public void inAYyyXxx(AYyyXxx node) { ... }
public void outAYyyXxx(AYyyXxx node) { ... }
public void caseAYyyXxx(AYyyXxx node) { ... }
```

構文木は名前通り深さ優先順でたどられますが、最初にそのノードに到達するときに in メソッドが呼ばれ、最後にそのノードから出ていくときに out メソッドが呼ばれ、その間で子ノードに対する apply() が呼ばれます。多くのノードはこの「最初」「最後」だけで用が足りるのですが、「途中」でも処理が必要な場合は case メソッドをオーバライドして使用します。ただし case メソッドをオーバライドした場合、その中で自分で子ノードの apply() を呼ばなければ、子ノードはたどられません (したがって、たどりたくない場合にも case をオーバライドします)。つまり、次のようにするのが標準です。

```
@Override  ←名前を間違えやすいので必ずこのアノテーションをつける
public void caseAYyyXxx(AYyyXxx node) {
  // 最初に到達したときの処理…
  node.getAa().apply(this);
  // aa と bb の間の処理…
  node.getBb().apply(this);
  // bb と cc の間の処理…
  node.getCc().apply(this);
  // 終って出て行くときの処理
}
```

さて、これでオーバライドのしかたは分かりましたが、あと 1 つ説明すべきことが残っています。構文木をたどりながら処理をするとき、ノード間でデータを受け渡していくのが普通ですが、メソッドの形は上のように決まっているので、受け渡すデータのためのパラメタを追加することができません。この問題に対処するため、SableCC では DepthFirstAdapter において、データの受け渡し用に、次のメソッドを用意しています。

```
void setIn(Node node, Object x);
Object getIn(Node node);
void setOut(Node node, Object x);
Object getIn(Node node);
```

ここで In 側は木の上側から葉に向かってデータを流すのに使い、Out 側は葉から上側に向かってデータを戻すのに使うという想定です。格納されるのは Object 値なので、適宜キャストが必要です (古い Java のコンテナのスタイル)。

では先の文法記述に対するコンパイラドライバ (前回とまったく同じ) とツリーインタプリタを示します。

```
package sem3;
import sem3.parser.*;
import sem3.lexer.*;
import sem3.node.*;
import java.io.*;
import java.util.*;

public class Compiler {
  public static void main(String[] args) throws Exception {
    Parser p = new Parser(new Lexer(new PushbackReader(
      new InputStreamReader(new FileInputStream(args[0]), "JISAutoDetect"),
        1024)));
    Start tree = p.parse();
    Executor exec = new Executor();
    tree.apply(exec);
  }
}

package sem3;
import sem3.analysis.*;
import sem3.node.*;
import java.io.*;
import java.util.*;

class Executor extends DepthFirstAdapter {
  Scanner sc = new Scanner(System.in);
  PrintStream pr = System.out;
  HashMap<String,Integer> vars = new HashMap<String,Integer>();
  @Override
  public void outAAssignStat(AAssignStat node) {
    vars.put(node.getIdent().getText(), (Integer)getOut(node.getExpr()));
  }
  @Override
  public void outAReadStat(AReadStat node) {
    String s = node.getIdent().getText();
    pr.print(s + "> "); vars.put(s, sc.nextInt()); sc.nextLine();
  }
  @Override
  public void outAPrintStat(APrintStat node) {
    pr.println(getOut(node.getExpr()).toString());
  }
  @Override
  public void caseAIfStat(AIfStat node) {
    node.getExpr().apply(this);
    if((Integer)getOut(node.getExpr()) != 0) { node.getStlist().apply(this); }
  }
  @Override
  public void caseAWhileStat(AWhileStat node) {
```

```
  while(true) {
    node.getExpr().apply(this);
    if((Integer)getOut(node.getExpr()) == 0) { break; }
    node.getStlist().apply(this);
  }
}
@Override
public void outATermExpr(ATermExpr node) {
  setOut(node, getOut(node.getTerm()));
}
@Override
public void outAGtExpr(AGtExpr node) {
  if((Integer)getOut(node.getLeft()) > (Integer)getOut(node.getRight())) {
    setOut(node, new Integer(1));
  } else {
    setOut(node, new Integer(0));
  }
}
@Override
public void outALtExpr(ALtExpr node) {
  if((Integer)getOut(node.getLeft()) < (Integer)getOut(node.getRight())) {
    setOut(node, new Integer(1));
  } else {
    setOut(node, new Integer(0));
  }
}
@Override
public void outAFactTerm(AFactTerm node) {
  setOut(node, getOut(node.getFact()));
}
@Override
public void outAAddTerm(AAddTerm node) {
  int v = (Integer)getOut(node.getTerm()) + (Integer)getOut(node.getFact());
  setOut(node, new Integer(v));
}
@Override
public void outASubTerm(ASubTerm node) {
  int v = (Integer)getOut(node.getTerm()) - (Integer)getOut(node.getFact());
  setOut(node, new Integer(v));
}
@Override
public void outAIdentFact(AIdentFact node) {
  setOut(node, vars.get(node.getIdent().getText()));
}
@Override
public void outANumberFact(ANumberFact node) {
  setOut(node, Integer.parseInt(node.getNumber().getText()));
```

```
    }
  }
```

基本的に、式の中では、それぞれの式の値を out メソッドで計算して setOut() でノードの出力値として保持します。中間のノードは子ノードの値を取って来て必要に応じて計算し、自ノードの値とします。read 文や print 文はその場でそれぞれの動作をします。if 文や while 文は、条件部をまず実行し、その結果に応じて本体部の実行を制御するわけです。

では、さっきのプログラムを実行してみます。

```
% cat sem3.txt
n に読み込む
x に 1 を入れる
n>0 の間 x に x+x を入れる n に n-1 を入れるを実行
x を打ち出す
% java sem3.Compiler sem3.txt
n> 8
256
%
```

演習 2 上の例をそのまま動かしなさい。動いたら、演習 1 で書いたプログラムを動かしてみなさい。その後、言語に次のような変更を行ってみなさい。

a. if 文に else 部が書けるように直して、その上で 2 数の最大」を動かしてみなさい。
b. do-while 文のような「下端で条件を調べるループ」を追加してみなさい。
c. その他、好きな変更を行ってみなさい。

演習 3 この方式で自分の好きな言語を設計して実装してみなさい。

2 ネットワークと分散システム

分散システム (distributed system) とは、物理的に複数の箇所に設置されている要素の集まりとして構成されているようなシステムを言います。電子メール、DNS(ドメイン名を管理しているシステム)、WWW などはすべて分散システムの例です。そもそもインターネット自体が巨大な分散システムだと言えます。

なお、「銀行の預金管理システム」「JR の座席予約システム」のようなものも複数の箇所に設置されていますが、あまり分散システムとは呼びませんね (どちらかというと「オンラインシステム」と呼ばれる)。これは、センターがほとんどの業務を行い、各地に分散しているのは入出力をこなすための「端末」だからでしょうか。

分散システムの利点は、それが「分散している」ところにあります。つまり、さまざまな箇所から「そこにある」システムを使うことができ、その結果が (システム全体の整合性として) 他の箇所と共有されます。つまり、「遠隔通信」「データの共有」「共同作業」などが実現できるわけです (図 1)。

今日、分散システムは非常に多くなっていますが、これはインターネットというインフラが整備され、その上に「ちょっと」固有の機能を載せるだけでさまざまな分散システムが作れるようになったことと、ネットの普及によりユーザが分散システムの便利さに慣れてきていて、新たな分散システムに対する抵抗がない (むしろ欲している) という側面があるからでしょう。

実は、今日もっとも多い分散システムの形態は「Web ベースのシステム」です。これは、ブラウザさえあればとくにソフトウェアをインストールしなくても使える (インストールレス) という面と、ユーザもブラウザは必ず使うので、システムに対する敷居が小さくなるという側面があります。

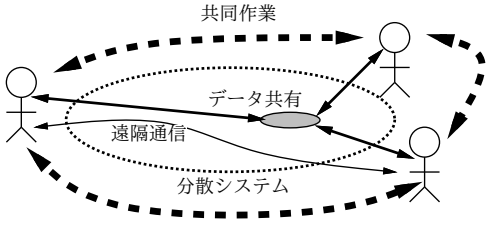

図 1: 分散システムが提供するもの

とはいえ、Web ベースのシステムの話はどこかよそで沢山やっていますし、オブジェクト指向技術的に面白いことはむしろそれ以外の部分にあるので、今回は「Web ベースでは無い」ような分散システムの話を中心に進めて行きます。

3 クライアントサーバ (C/S)

3.1 C/S の原理、利点/欠点

今日の最も標準的な分散システムのモデルは、クライアントサーバ (C/S) モデルだと言えます。C/S モデルは、次のようになっています。

- システムは、サービスを提供するサーバと、サービスを利用するクライアントから成る。小規模なサービスではサーバは 1 つということもある。クライアントは複数。
- サーバは常時稼働していて、ネット上の決められた (知られている) 場所 (ホスト名+ポート番号) でサービスを受け付けている。
- クライアントはユーザがサービスを利用したいときに立ち上げる。クライアントはサーバに接続し、サーバとやりとりして、そのユーザのための機能を提供し、用が済んだら接続を終了。

なぜこうなっているかというと、プログラム A と B が通信するためには、A と B がともに動いている必要があるわけで、そのための一番簡単で明らかな方法はどちらか (たとえば B) をずっと動かしておくこと、だからだと言えます。

これとも重複しますが、C/S 構成の利点は次の点でしょうか。

- 共有のデータはサーバ上にずっと保持することができ、これにより情報共有の基本的な機能が提供できる。
- 共有データを保持するのがサーバ上だけであれば、データの管理 (セキュリティ対策、整合性の管理等、運用管理) がやりやすい。
- クライアントは各ユーザの都合で UP/DOWN していいので、ユーザにとっても使いやすい。

これらの利点のため、C/S 構成はものすごく古いにもかかわらず、今日のネットワークサービスにおいても主流であり続けています。Web も Twitter も Facebook も皆「サーバ」がサーブしていますよね。一方、C/S には次のような弱点があります。

- サーバがすべてのやりとりを仲介するので、ボトルネックになりやすいし、単一故障点 (single point of failure) にもなりやすい。
- スケーラビリティが実装しにくい。

これらの弱点を回避する方法としては、サーバを多数用意して振り分けることですが、そうするとそれらのサーバ間でどのように情報を共有するかということが難しくなります。サーバ群の中で P2P(複数のシステムが平等な立場で情報をやりとりする方式) 的にデータをやりとりすることで共有するなど、ややこしい方法が必要です。

3.2 簡単なC/Sプログラム

とりあえず、あんまりオブジェクト指向ではないのですが、簡単なC/Sプログラムを書いて動かしてみましょう。まあ、通信のためのソケットやストリームをオブジェクトとして作ったり受け取ったりするというところはオブジェクト指向ならではのやりやすさだと言うこともできます。まず、クライアント側から見てみましょう。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.net.*;
import java.util.*;

public class Sample41 extends JPanel {
  JTextField f1 = new JTextField();
  JButton b1 = new JButton("Exec");
  JLabel l1 = new JLabel("start...");
  public Sample41(final String s) {
    setLayout(null);
    add(f1); f1.setBounds(40, 40, 180, 40);
    add(b1); b1.setBounds(240, 40, 80, 40);
    add(l1); l1.setBounds(20, 360, 360, 40);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          Socket cs = new Socket(s, 4192);
          PrintWriter out = new PrintWriter(cs.getOutputStream(), true);
          Scanner sc = new Scanner(cs.getInputStream());
          out.println(f1.getText());
          f1.setText(sc.nextLine());
        } catch(Exception ex) { l1.setText("!" + ex); }
      }
    });
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sample41(args[0]));
    app.setSize(400, 400);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

あんまり詳しく説明すると大変なのでおおまかに説明します。

- `main()` と当該クラスのについてはこれまでのと同じ。ただし、コマンド引数の先頭のものをコンストラクタへのパラメタとして渡すようにしている (サーバのホスト名を指定したいから)。
- `Sample41` は例によって `JPanel` のサブクラスで窓にはめるが、今回はその中に「入力欄」「ボタン」「表示欄」を1つずつ配置する。

- Exec ボタンが押されると、その動作の中で、指定したサーバのポート 4192 に接続し、入力欄の文字列を送出し、結果を 1 行読んで入力欄に書き換え表示する。途中で例外が起きたらそれは表示欄に表示する。

非常に簡単ですね。なお、ソケットというのはネットワーク通信 (TCP 接続) の端点に対応します。
では次にサーバ側のコードを示します。

```
import java.net.*;
import java.io.*;
import java.util.*;

public class Sample41Server {
  public static void main(String[] args) throws Exception {
    ServerSocket ss = new ServerSocket(4192);
    System.out.println("starting...");
    while(true) {
      Socket cs = ss.accept();
      PrintWriter out = new PrintWriter(cs.getOutputStream(), true);
      Scanner sc = new Scanner(cs.getInputStream());
      String line = sc.nextLine();
      out.println(line + line);
      cs.close();
      if(line.equals("bye")) { break; }
    }
    ss.close();
  }
}
```

こちらはもっと簡単です。サーバソケットを用意し、繰り返し、接続を待って、クライアントが接続してきたら文字列を 1 行読み込み、その文字列を 2 回くつつけた文字列を返します。なお、渡された文字列が bye だったらなぜかサーバは終わります。簡単ですね?

演習 4 サーバとクライアントのプログラムを持って来てそのまま動かしなさい (2 つ窓が必要です。また、クライアント側を動かす時にはホスト名を指定すること)。動いたら、サーバ側を次のように変更してみなさい。

a. 受け取った文字列をサーバ側の画面に表示する。
b. クライアントに返す文字列を変更してみなさい。
c. 「空文字列が来たら覚えている文字列を返し、空でない文字列が来たらその文字列を覚える」ようにしてみなさい。完成したら、複数人でチャットしてみなさい。

4 RPC と RMI

4.1 RPC(遠隔手続き呼び出し)

先のような C/S プログラムでは、通信路を張ってサーバとクライアントで往復しつつ処理を進めるので、その「やりとりのしかた」(プロトコル) を設計し実装するのが面倒です。たとえば telnet とか SMTP とか、インターネットの初期からあるプロトコルは文字ベースですが、たとえば次のような感じでできています。

```
% telnet utogw smtp  ←SMTP ポートを指定してメールサーバに接続
Trying 192.xx.xx.x...
Connected to utogw
Escape character is '^]'.
220 gssm.otsuka.tsukuba.ac.jp ESMTP
MAIL FROM:<kuno>  ←自分が誰かを示す (エンベロープ From)
250 ok
RCPT TO:<kuno>  ←メール宛先を示すコマンド
250 ok
DATA          ←「以下本文」コマンド
354 go ahead
From: kuno    ←メールヘッダが最低 1 つは必要
              ←空っぽの行がヘッダ終わりを示す。
test...       ←本文も 1 行以上あった方がよい。
.             ←「.」だけの行があるとおしまいを表す。
250 ok 989909466 qp 19829
QUIT          ←「これでおしまい」コマンド
221 gssm.otsuka.tsukuba.ac.jp
Connection closed by foreign host.
%
```

手で打つぶんにはこれでもよさそうかも知れませんが、これをプログラムでやりとりするとなると面倒そうだと思いませんか? ネットワークを「たまに」使うだけならこれでもいいかも知れませんが、今日のようにそこら中がネットワークと分散だらけになると、そのたびにこういうものを実装するのでは大変すぎます。

そこで、もっとプログラム的に楽な、扱いやすいやりとりの仕方が模索されました。その 1 つが RPC(遠隔手続き呼び出し) です。その基本的なアイデアは、次のことがらです。

- 手続き呼び出しは多くのプログラミング言語において基本的な構成要素であり、プログラマはそれを使うことに慣れている。
- 手続き呼び出しを、それと同様に動作するネットワーク経由の通信に変換できる。

後者についてもう少し説明しましょう。手続き呼び出しは、呼び側のコードで呼び出し命令を実行すると、制御が呼ばれた手続きに移り、手続きの中を実行して最後に戻り命令を実行すると、呼び出し命令の直後の箇所に戻って実行を続けます (図 2 左)。手続き内を実行している間は、呼び側の実行は停止しています (1 つの CPU で実行しているのだから当然ですが)。

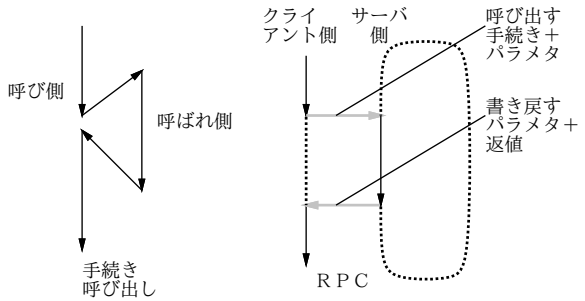

図 2: 手続き呼び出しと遠隔手続き呼び出し

これをネットワークに広げたものが遠隔手続き呼び出し (RPC、remote procedure call — 図 2 右) です。RPC ではクライアント側で動くコードが呼び側となり、呼び出しの箇所で「どの手続きを呼

ぶか」「パラメタの値は何か」という情報をサーバ側に伝えます。サーバはこれらの情報を受け取ると、対応する手続きの処理を実行し、結果を返送します (返値だけのこともありますし、パラメタを書き換える場合は書き換える値も送ります)。なお、呼び側 (クライアント側) は送信から受信までの間は「(返答を) 待っている」状態、また呼ばれ側 (サーバ側) は呼ばれるまでと呼ばれた後は「(呼び出しを) 待っている状態」であることに注意。

これの何がいいのかというと、呼び側にとっては「単に手続きを呼んでいる」だけに見えますし、呼ばれ側にとっても「単に手続きとして呼ばれているだけ」に見えるので、普通のプログラムを書いているかのような気持ちで開発ができるという点です (実際にはちょっと違う点があります…後述)。

しかし、呼んでいるだけ、呼ばれているだけといっても全然違うじゃないか、と思うかも知れませんが、実際に呼び側も呼ばれ側もたたの手続きとして記述することができます。それには次の方法を使います (図 3)。

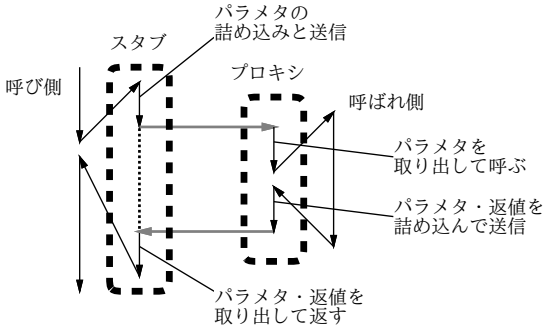

図 3: RPC のスタブとプロキシ

- 呼び側は、本当の呼ばれ側を直接呼ぶかわりに、同じマシン上の「抜けがら」(スタブ) を呼ぶ。
- スタブはサーバ側と通信してパラメタなどの情報を渡す。
- サーバ側では「代理」(プロキシ) がパラメタなどの情報を受け取ってメモリ上に置き、それをパラメタとして本物の呼ばれ側を呼ぶ。
- 呼ばれ側は普通の手続きなので普通に仕事をして返値を返す (パラメタも書き換えるかも)。
- プロキシは返値と書き換えられたパラメタをネットワーク経由で返送する。
- スタブは受け取ったパラメタの書き換えを行い、返値を返す。
- 呼び側は普通に手続き呼び出しから戻って来たので仕事を続行。

ここで、スタブとプロキシは「単にデータを受け渡す」だけが仕事なので、手続き名とパラメタの個数や型さえわかっていれば、機械的に生成できます。この生成ツールをスタブジェネレータと呼びます。つまり、スタブジェネレータと予め用意した汎用のサーバがあれば、普通の手続き呼び出しの呼ばれ側をサーバ側で動くように変換することは簡単に行えるわけです。

なお、パラメタをネットワーク経由で送るには、色々な手間が必要です。たとえば整数型ですら、マシンによってバイト順が違うために「中立な」表現で送る必要がありますし、ポインタをそのまま送っても意味がないので、レコードやポインタを含むデータ構造は「たどりながら全部詰めて行く」「詰めたものからもとの構造を復元する」という処理が必要になります。これを詰め込み (marshalling)、取り出し (demarshalling) と呼びます。

スタブの生成、詰め込み/取り出しコードの生成、プロキシをロードできる汎用のサーバ、保護のための認証機構などの機能一式は PRC システムとしてパッケージされて開発されるのが普通です。たとえば FreeBSD、Solaris などでは ONC RPC と呼ばれるパッケージが使われていて、「`rpcinfo`」でどのような遠隔手続きが登録され利用可能かを調べることができます。

4.2 RMI(遠隔メソッド起動)

ここまでは「手続き」の話でしたが、オブジェクト指向の普及とともに、呼び出される対象は「オブジェクトのメソッド」と考えるのが良さそうになってきました。つまり、オブジェクトは固有のデータを中に保持しているので、そのオブジェクトが「どこか」にあって、それを「さまざまなメソッドを呼ぶ」ことで利用する、という形が分かりやすいわけです。これに対応して、メソッド呼び出しを受け付けて振り分けるサーバ部分のことを ORB(Object Request Broker) と呼ぶようになりました。

ORB としては先駆的な国産の HORB (Hirano ORB) をはじめ色々なものが作られましたが、相互運用性のためにさまざまな言語、システムをまたがった共通のものを作ろうという動きが生まれ、OMG(Object Management Group) が設立されて CORBA(Common Request Broker Architecture) が制定されました。CORBA で定められているものには、呼び出せるオブジェクトのメソッド名やパラメタを記述する言語 IDL(Interface Definition Language — この記述をスタブジェネレータの入力にしたり、人間が仕様を参照するのに使う)、呼び出しに使うプロトコルなど多くのものがあります。

CORBA は仕様であり、これに従っているなら、どの言語で書かれたどのクライアントからでもどの言語で書かれたどのサーバオブジェクトも呼べることになっています。が、そのような汎用性のために複雑かつ遅くなり、あまり成功したとは言えません。今日では、遠隔呼び出しは Web サービスを直接叩くことが普通になり、そのために SOAP(Simple Object Access Protocol) や XML-RPC など XML ベースの技術が使われるようになっています。そして OMG はこれらの技術や UML などオブジェクト指向関係のさまざまな仕様を制定し管理する組織として発展しています。

一方、Java ももともと分散を指向したシステムだったので、Java 言語内部で独自の ORB を提供し、その枠内であれば簡単に使えるようになっています。これは一般に Java RMI と呼ばれています (RMI は Remote Method Invocation、つまり遠隔メソッド起動の意味)。なお、CORBA と接続するための機能ももちろんありますが、そちらは例によって複雑で面倒ですし、ここでは扱いません。

4.3 例題: Java RMI によるメソッド呼び出し

Java RMI では、遠隔呼び出しされるオブジェクトに対してそれが従うインタフェースを用意し、そのインタフェースにあるメソッドだけが他のホストから呼ばれるようにしています。ここでは例題として、非常に簡単なメソッド `add()`(覚えている数値に渡された値を加え、合計を返すものとします) だけを持つインタフェースを用意しました。

```
import java.rmi.*;

interface Sample42IF extends Remote {
  public int add(int i) throws RemoteException;
}
```

このインタフェースに対しては後からスタブを生成すべきなので、その印として `extends Remote` と指定します。また、普通のメソッド呼び出しならエラーは (呼び出しそのものに関しては) 出ないはずですが、遠隔呼び出しだとネットワークが切れていたり色々起きるので、それを表す例外 `RemoteException` を投げるものと宣言します。

次はサーバ側オブジェクトですが、上記のインタフェースに従い、またサーバ側として動作する機能をクラス `UnicastRemoteObject` から継承したクラスを作ります。ここではコンストラクタ (何もしませんが、やはり遠隔呼び出し関係の失敗が起きることを表すため `RemoteException` を投げるとしています)、遠隔側から呼ばれるメソッド `add()`、そしてローカルにだけ使うメソッド `get()` を持ちます。

```
import java.rmi.*;
import java.rmi.server.*;
```

```
import java.rmi.registry.*;
import java.util.*;

public class Sample42Server extends UnicastRemoteObject implements Sample42IF {
  int data = 0;
  public Sample42Server() throws RemoteException { }
  public int add(int i) { return data += i; }
  public int get() { return data; }

  public static void main(String[] args) throws Exception {
    Scanner sc = new Scanner(System.in);
    Sample42Server srv = new Sample42Server();
    Registry reg = LocateRegistry.createRegistry(4918);
    reg.bind("add", srv);
    System.out.print("command> ");
    while(!sc.nextLine().equals("bye")) {
      System.out.println(srv.get());
      System.out.print("command> ");
    }
    System.exit(0);
  }
}
```

メソッド main() はサーバを起動し (これは LocateRegistry.createRegistry() にポート番号を指定することで行えます)、そのサーバにサーバ側オブジェクトを「add」という名前で登録します。あとは、1 行入力されるごとに、現在保持している値を表示します。サーバ側をコンパイルし、続いてスタブ/プロキシを生成し、サーバをさっさと起動します。

```
% javac Sample42IF.java
% javac Sample42Server.java
% /compat/linux/usr/local/Java/jdk1.6.0_22/bin/rmic Sample42Server
% java Sample42Server
command>
```

では、クライアント側を見てみましょう。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.rmi.*;

public class Sample42 extends JPanel {
  Sample42IF srv;
  JTextField f1 = new JTextField();
  JButton b1 = new JButton("Exec");
  JLabel l1 = new JLabel("start...");
  public Sample42(String url) throws Exception {
    srv = (Sample42IF)Naming.lookup(url);
    setLayout(null);
    add(f1); f1.setBounds(40, 40, 180, 40);
```

```
    add(b1); b1.setBounds(240, 40, 80, 40);
    add(l1); l1.setBounds(20, 360, 360, 40);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          f1.setText(""+srv.add(Integer.parseInt(f1.getText())));
        } catch(Exception ex) { l1.setText("!" + ex); }
      }
    });
  }
  public static void main(String[] args) throws Exception {
    JFrame app = new JFrame();
    app.add(new Sample42(args[0]));
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setSize(400, 400);
    app.setVisible(true);
  }
}
```

`main()` 側はこれまでと同じですが、コマンド引数で rmi: URL 形式でサーバとオブジェクト名を指定します。この URL は次の形を取ります。

`rmi://`ホスト名`:`ポート`/`登録名

先にポート番号は 4918、登録名は add にしたので、次のように起動します (ホスト名はサーバを動かしているマシンを指定)。

```
% java Sample42 rmi://sma:4918/add
```

図 4: Sample42 の画面

このサーバに複数箇所から接続して呼び出すことで、「全員の値を合計」のようなことが行えます。

演習 5 例題をコピーしてきて、そのまま動かしなさい。動いたら、次のことをやってみなさい。

a. 誰かのサーバ 1 つを決めて、全員で頭に思い描いた数字を一斉に送り、正しく合計されたことを確認しなさい。

b. `add()` の中身を別の処理に変更して、他の人にサーバに接続してもらい、どのような処理にしたかあててもらいなさい。

5　オブジェクトのモビリティ

5.1　オブジェクトの移送とシリアライズ

モビリティ(mobility) とは、ネットワークを経由して「何か」が移動できることを言います。たとえば先の RPC でもネットワークを経由してパラメタなどが転送されていました。このとき、構造のあるデータのコピーは必ずしも簡単でない、という話題があります。先ず、図 5 のようにポインタでつながったデータの場合、ポインタをそのまま他のマシンにコピーしてもあらぬ場所を指してしまって意味がないので、ポインタの指す先を併せて転送し、転送後にそれらを指すポインタで元のポインタを置き換える必要があります

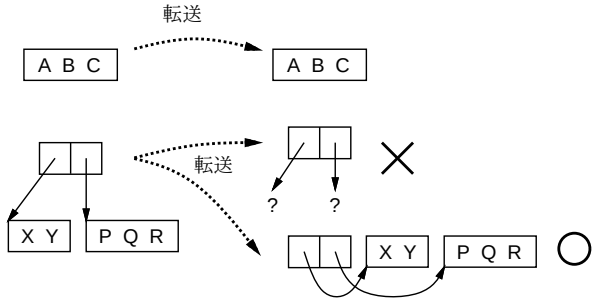

図 5: ポインタを含んだデータのコピーの問題

さらにややこしいのは、図 6 のように構造の一部が共有されている場合、その構造をそのまま維持してコピーする必要があります。この処理は、ガベージコレクション (GC) と同様にコピーするデータ構造をたどりながら印をつけながら転送し、1 回印をつけたものは 2 回目は転送せずに行き先で 1 回目のものと同じ場所を指すようにポインタをつけることで行えます。

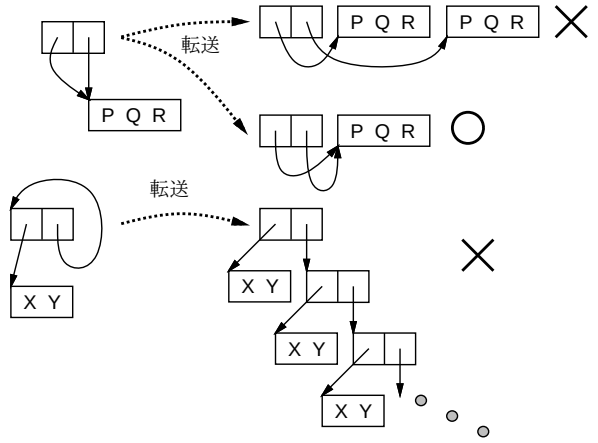

図 6: 共有や再帰を含んだデータのコピーの問題

Java では、クラスにおいて「`implements Serializable`」を指定すると、そのクラスのオブジェクトはシリアライズ可能 (移送可能) になります。その場合、このクラスのインスタンスはストリームに書き込んでバイト列に変換 (詰め込み) でき、ネットワーク経由で送った後、バイト列から元の状態が復元 (取り出し) できるようなコードが自動で用意されます (そのためには、そのクラスや親クラスすべてが引数なしのコンストラクタを持ち、またインスタンス変数群すべてがまた `Serializable` であるか基本型である、などの条件が必要となります。このとき、先に述べたようなポインタの変換なども自動で対処されます。なお、この機能による自動的な詰め込み/取り出しでは不都合な場合 (上

記の条件が満たせないなどの場合) のために、もっと細かい制御をしながら読み書きする機能も別に用意されています。

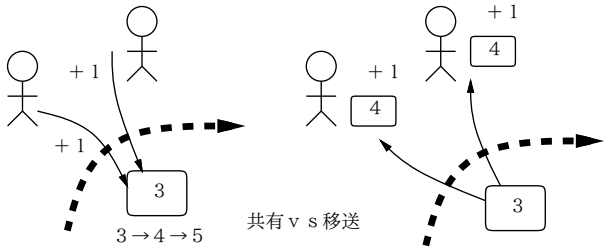

図 7: 共有と移送の違い

ところで、分散オブジェクトのシステムの場合、遠隔サイトにあるオブジェクトへのアクセスが「遠隔参照 (共有セマンティクス)」なのか「移送 (コピーセマンティクス)」なのかという問題があります。たとえば、先の RMI の場合は、遠隔オブジェクトのメソッドを呼び出していたので、オブジェクトは遠隔参照され、複数箇所で共有されていました。これに対し、移送のモデルではオブジェクトは手元に持ってこられるので、複数箇所に持って来られたオブジェクトは別物です (図 7)。

Java では、Remote インタフェースを使ってアクセスするオブジェクトは遠隔参照になり、Serializable インタフェースを持ったオブジェクトのやりとりは移送になるので、両方の機能があるわけですが、これを使い分けるというのはやや混乱しそうでもあります。もっと統一的に「すべてのオブジェクトは遠隔参照」のようにするシステムも研究されてきましたが、オブジェクトの管理や分散ごみ集めなどに難しさがあり、また性能上も不利なところがあって、普及していません。

5.2 例題: シリアライズによるオブジェクトのやりとり

では具体的な例題として、シリアライズ可能なオブジェクトをサーバとやりとりしてみます。今回やりとりするオブジェクトは、内部に文字列のリストを保持し、1 行ずつの追加と各行の取り出しが行える、というものです。

```
import java.io.*;
import java.util.*;

public class Sample43Data implements Serializable, Iterable<String> {
  ArrayList<String> list = new ArrayList<String>();
  public void add(String s) { list.add(s); }
  public Iterator<String> iterator() { return list.iterator(); }
}
```

サーバとの RMI インタフェースはこのオブジェクトを「取る」のと「戻す」のと 2 つのメソッドを持たせます。

```
import java.rmi.*;

interface Sample43IF extends Remote {
  public Sample43Data get() throws RemoteException;
  public void put(Sample43Data d) throws RemoteException;
}
```

サーバはこれを実装し、取ったときは中は空っぽになり、戻されるとそれを保持します。これにより、1 つのサーバについては 1 つだけそのオブジェクトがあるわけです。あと、印刷用に「覗いて取り出す」メソッドを別途用意しました。

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.util.*;

public class Sample43Server extends UnicastRemoteObject implements Sample43IF {
  Sample43Data data = new Sample43Data();
  public Sample43Server() throws RemoteException { }
  public Sample43Data get() { Sample43Data d = data; data = null; return d; }
  public void put(Sample43Data d) { data = d; }
  public Sample43Data peek() { return data; }

  public static void main(String[] args) throws Exception {
    Scanner sc = new Scanner(System.in);
    Sample43Server srv = new Sample43Server();
    Registry reg = LocateRegistry.createRegistry(4918);
    reg.bind("list", srv);
    System.out.print("command> ");
    while(!sc.nextLine().equals("bye")) {
      if(srv.peek() != null) {
        for(String s: srv.peek()) { System.out.println(s); }
      }
      System.out.print("command> ");
    }
    System.exit(0);
  }
}
```

最後にクライアントですが、今度はボタンを 2 つに増やして、「サーバから取る/サーバへ戻す」ボタンと「手元に持って来たオブジェクトに対して文字列を追加する」ボタンを持たせました。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.rmi.*;

public class Sample43 extends JPanel {
  Sample43IF srv;
  Sample43Data data;
  JTextField f1 = new JTextField();
  JButton b1 = new JButton("Add");
  JButton b2 = new JButton("Get/Put");
  JLabel l1 = new JLabel("start...");
  public Sample43(String url) throws Exception {
    srv = (Sample43IF)Naming.lookup(url);
```

```
    setLayout(null);
    add(f1); f1.setBounds(20, 40, 150, 30);
    add(b1); b1.setBounds(200, 40, 80, 30);
    add(b2); b2.setBounds(300, 40, 80, 30);
    add(l1); l1.setBounds(20, 260, 360, 40);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          if(data == null) {
            l1.setText("data is not available");
          } else {
            data.add(f1.getText()); l1.setText(""); f1.setText("");
          }
        } catch(Exception ex) { l1.setText("!" + ex); }
      }
    });
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          if(data == null) {
            data = srv.get();
            l1.setText((data==null) ? "unavailable" : "success");
          } else {
            srv.put(data); data = null; l1.setText("released");
          }
        } catch(Exception ex) { l1.setText("!" + ex); }
      }
    });
  }
  public static void main(String[] args) throws Exception {
    JFrame app = new JFrame();
    app.add(new Sample43(args[0]));
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setSize(400, 300);
    app.setVisible(true);
  }
}
```

このようなモデルは、次のような利点があることに注意。

- オブジェクトは「1 つ」なので、それを自分が持っていれば他人がそれに干渉することはない (トークンのような役割)
- ローカルに持って来て操作するので、操作を大量に行っても能率がよい。

一方で次のような弱点もあります。

- オブジェクトの状態全部を移送するので、状況によってはネットワーク帯域が沢山必要になる
- 他人が持っている間は使えないので待たなければならない (利点と表裏)

- 誰かが「無くして」しまうと永遠に失われる (控えを用意したり無くなった時に回復するなどの手段が本来は必要)

演習 6 この例題を持って来てそのまま動かしてみなさい。動いたら次のようなことも試してみなさい。

a. 誰かのサーバのデータに皆で一斉に 2 行ずつメッセージを書き込もうとしてみる。

b. この例題を改造して何かもっと面白いことをさせる。

6 モバイルエージェント

せっかくここまで来たので、モバイルエージェントのおもちゃを作ってみることにします。モバイルエージェントとは、「オブジェクトが自律的にネットワーク中を移動しつつ、固有のタスクを行う」ようなものを言います。エージェントの用途としては、たとえば次のようなものが考えられます。

- ネットワークのあちこちに行って情報を収集したり配布する。
- CPU のあいているマシンに行って定められた計算をする。
- ユーザに指示された仕事をあちこちを回って順次こなす (その間ユーザは自分のマシンを落としていてもよい)

ここで重要なのは、エージェントは「自律的に」動く、つまり何をしてほしいかはプログラムをした (作った) 人が決めるにせよ、その後実際に実行を始めたら作った人とは離れて自分で行き先を決めながら動く、ということです。

なお、勝手に動くといっても、動くためには CPU が必要ですし Java であれば JVM が必要です。そこで、エージェントを動かすための「土台」(プラットフォーム) を用意し、エージェントはこれに乗っかってその場所での自分の処理を実行し、終ったら次の場所に自分を移動させることを繰り返すわけです。

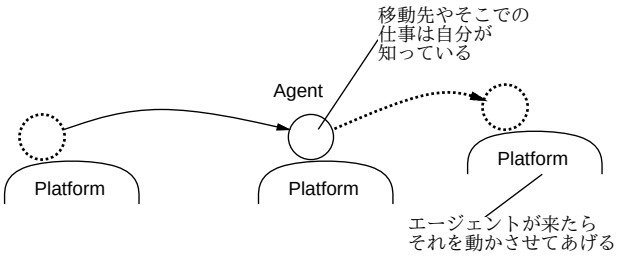

図 8: モバイルエージェントの概念

では実際に、簡単なエージェントのデモを作ってみましょう。まず RMI インタフェースの側から。

```
import java.rmi.*;

interface Sample44IF extends Remote {
  public void arrive(Sample44Agent d) throws RemoteException;
  public void leave(String s) throws RemoteException;
}
```

つまりエージェントは到着し、仕事が終るとメッセージを残して去って行くとうい感じです。ではこれを実装するサーバを見てみます。

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.util.*;

public class Sample44Server extends UnicastRemoteObject implements Sample44IF {
  Sample44Agent agt = null;
  public Sample44Server() throws RemoteException { }
  public void leave(String s) { System.out.println(s); }
  public void arrive(Sample44Agent a) { agt = a; a.exec(this); }

  public static void main(String[] args) throws Exception {
    Scanner sc = new Scanner(System.in);
    Sample44Server srv = new Sample44Server();
    Registry reg = LocateRegistry.createRegistry(4918);
    reg.bind("agent", srv);
    System.out.print("command> ");
    while(!sc.nextLine().equals("bye")) {
      System.out.print("command> ");
    }
    System.exit(0);
  }
}
```

実は到着したらただちにその到着したエージェントの exec を呼んでいるだけです。さて、実はこの Sample44Agent というのもインタフェースです。

```
import java.io.*;

interface Sample44Agent extends Serializable {
  public void exec(Sample44IF home);
}
```

なぜそうしたかというと、この後演習でさまざまなエージェントを作ってみて頂きたいからです。とりあえず作ってみた簡単なエージェントは次のものです。

```
import java.io.*;
import java.util.*;
import java.rmi.*;

public class Sample44AgentImpl implements Sample44Agent {
  String[] hosts;
  int count = 0;
  public void setHost(String[] h) { hosts = h; }
  public void exec(Sample44IF home) {
    try {
      Thread.sleep(1000);
      if(home != null) { home.leave(++count + "th trip, bye!"); }
      String h = hosts[(int)(Math.random()*hosts.length)];
      String url = "rmi://" + h + ":4918/agent";
```

```
        Sample44IF srv = (Sample44IF)Naming.lookup(url);
        srv.arrive(this);
      } catch(Exception ex) { }
    }
    public static void main(String[] args) {
      Sample44AgentImpl agt = new Sample44AgentImpl();
      agt.setHost(args);
      agt.exec(null);
    }
  }
```

main() の側から読むと、エージェント実装を作って、どことどこを回るかのホストのリストを設定します。たとえば次のようにするわけえです。

```
% java Sample44AgentImpl ホスト1 ホスト2 ホスト3
```

そしてすぐ exec() を呼びます。さてその中ですが、まず 1 秒待ち、それから「さよなら」メッセージをもしプラットフォームが空でないなら…main() から呼ばれた時は空です…メッセージを表示します。続いて、ホストの並びから次の行き先をランダムに選び、そこに接続して、自分を送り込みます。これで、1 秒ごとに次々と放浪するエージェントができたわけです。

演習 7 例題を打ち込んでそのまま動かしてみなさい。そのとき、誰のエージェントか分かるようにメッセージを変更し、またクラス名もごっちゃにならないように変えなさい (クラス名を変えるとファイル名も対応して変える必要があることに注意。動いたら「時々自殺する」「時々分裂する」などの機能を持たせてみなさい。

演習 8 エージェントのおもちゃを改造して、何か面白いことをさせてみなさい。

ところで、上の演習をやってみて、不思議に思ったことがあるはずです…それは、自分のところで作ったエージェントが他人のプラットフォームで動くことです。不思議だと思いませんか? 他人のところにはあなたの書いたコード (を class ファイルに変換したもの) は無いはずですよね?

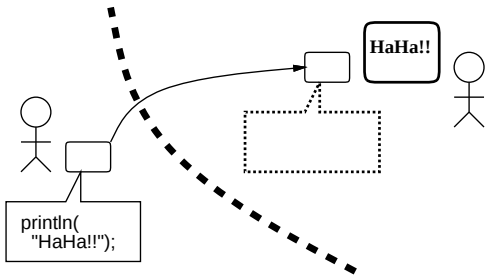

図 9: コードモビリティ

実は Java RMI では、オブジェクトを移送するとき、向う側にそのオブジェクトのクラスファイルが無ければ、クラスファイルも一緒に移送してくれます。だから、他人が作ったコードがやってきて勝手に動くわけです (図 9)。これを「コードモビリティ」といいますが、Java のような仮想マシン環境ならではの機能だとも言えます。

ただしこれは、他人が書いたコードが自分のマシンで動くことになるので、セキュリティ上の問題になる可能性が (もちろん) あります。今回は実験でおもちゃなのでいいのですが、まじめに実装する場合はきちんと対処方法を調べて設計してください。