オブジェクト指向技術'11 ＃5 — 分散/並列とOO

久野　靖*

2011.5.26

1 RPCとRMI

1.1 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。

*経営システム科学専攻

- 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 1 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)。

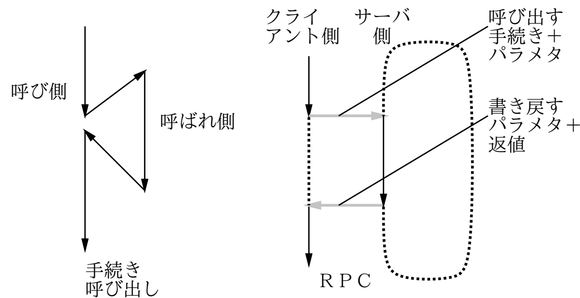

図 1: 手続き呼び出しと遠隔手続き呼び出し

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 1 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 2)。

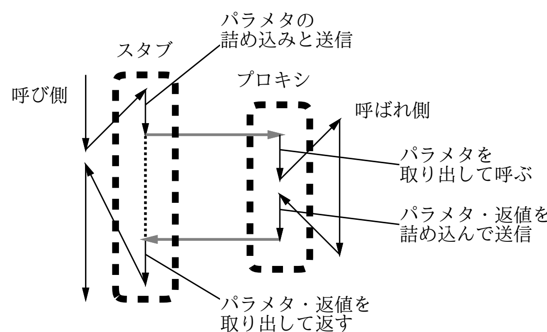

図 2: RPC のスタブとプロキシ

- 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。

1.2 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。

1.3 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
```

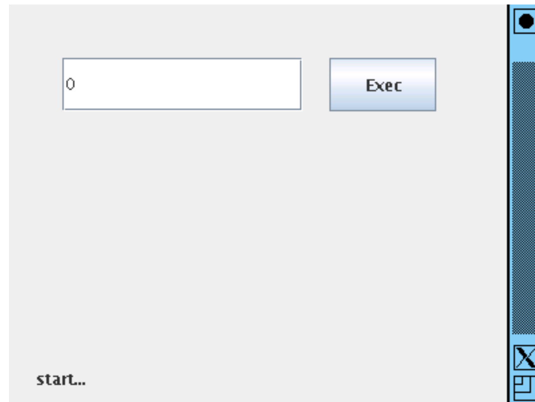

図 3: Sample42 の画面

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。

2 オブジェクトのモビリティ

2.1 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 4 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

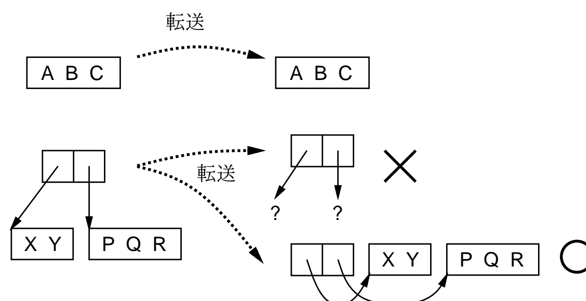

図 4: ポインタを含んだデータのコピーの問題

さらにややこしいのは、図 5 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

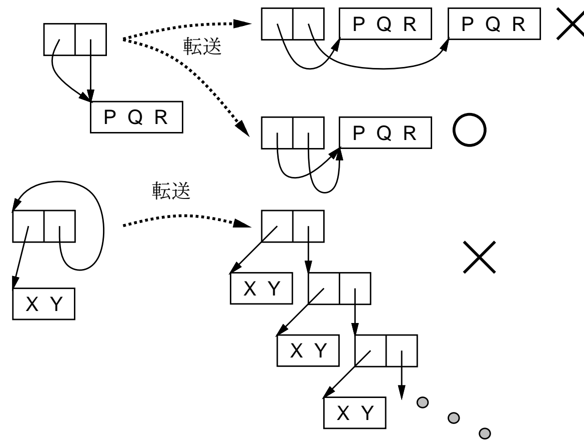

図 5: 共有や再帰を含んだデータのコピーの問題

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。

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。

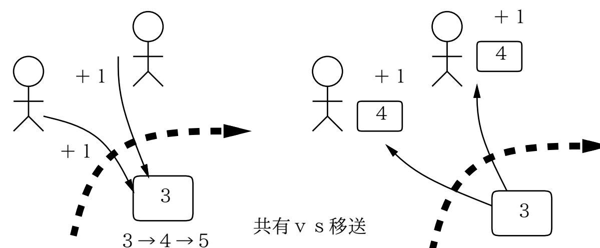

図 6: 共有と移送の違い

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 6)。

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。

2.2 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。

2.3 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

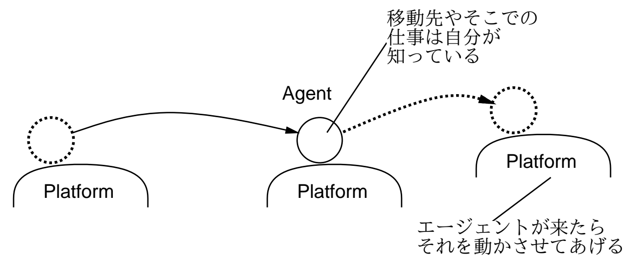

図 7: モバイルエージェントの概念

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。

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?

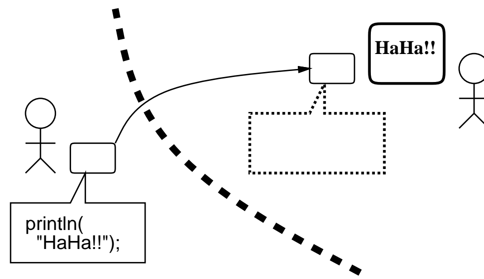

図 8: コードモビリティ

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 8)。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。

3 Linda (タプルスペース)

3.1 Linda の概念

最後に、少し毛色の違う通信モデルである Linda を紹介します。これは、1980 年代半ばころに David Gelenter が提唱した通信モデルで、Linda というのは言語の名前だったのですが、言語そのものはあまり重要でなく、その通信のフレームワークが非常に特徴があったので、この通信フレームワークの名前として Linda という名前が使われるようになりました。その概要は次のようなものです (図 9)。

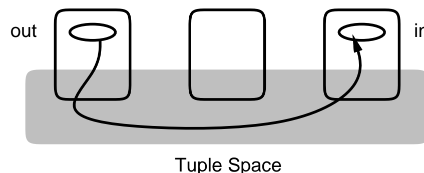

図 9: Linda とタプルスペース

- すべてのノードにまたがった、タプルスペース (tuple space、TS) と呼ばれる「場」が存在している。
- 各ノードは、TS にタプルを出したり、TS からタプルを取り込む形で通信する。
- TS に出されたタプルはあくまでも 1 個のタプルであり、勝手に増えたり減ったりすることはない (一種のトークンとして働く)。
- どのノードでも、TS からタプルを取り出そうとしたら、そのタプルが TS のどこかにあるなら (そして他のノードに先に取られなければ)、そのタプルを取り出すことができる。

- タプルには数値や文字列のような基本データが付随させられる。

具体的には、タプルは「名前」と複数の値を組にしたもので、出す時はその名前と値の組が出されるだけですが、取り込む時にはパターンマッチ機能により「この名前で、この値とこの値を持つもの」のような指定で取り出すことができます。
具体的なタプルに対する操作は次の 3 つがあります。

- `out` 名前 (値，値，…) — 指定した名前と値を組にしたタプルを TS に投入する。
- `in` 名前 (パラメタ，パラメタ，…) — 指定した名前のタプルを TS から取り込む。各パラメタは「`?`変数名」または値を指定する。パラメタが変数のときは、その変数にタプルの対応する値が取り出せる。値を指定したときは、対応するタプルも同じ値を持つものだけが取り込み対象になる。取り込まれたタプルは TS から無くなる。
- `read` 名前 (パラメタ，パラメタ，…) — `in` とほぼ同じだが、ただしタプルを読むだけで、TS から取り除くことはない。

このモデルの興味深いところは、次の点です。

- TS は全ノードにまたがっているので、どこで投入されたものでもどこからでも取り出せ、従って位置を意識しない通信が可能になる — 位置の分離。
- TS に入れたものはずっと残っているので、受信したい人は後のいつの時点にでもそれを取り出すことができる — 時点の分離。
- TS から取り出したタプルは無くなるので、同じものが複数あることはなく、整合性が維持できる。
- TS は多くのノードにまたがった分散サービスとして実装でき、これによってスケーラブルな実装が可能になる。

たとえば、これを使って次のようなサービスを簡単に実装することができます。

時刻サーバ

すべてのノードが正しく時間を知るのは簡単ではないですが、次のような時刻サーバを使うことが考えられます。

```
forever do
  out time(現在時刻の値)
  // 次の時刻まで待つ
  in time(?x)
end
```

時刻が次に進むと前の `time` タプルを削除して新しい時刻に取り替えます。これにより、どのノードからでも「`read time(?t)` で現在時刻を知ることができるわけです。

カウンタ

重複のない一連番号を割り振るというのも簡単ではない仕事です。これも、たとえば「`out count(1)`」を最初に実行しておき、あとは次の手順で番号を 1 つずつ取得できます。

```
countup:
  in count(?x)
  x = x + 1
  out count(x)
```

RPC

RPC も TS を使うと簡単に実装できます。サーバ側は次のようにします。

```
forever do
  in proc(?ret, ?para1, ?para2, ...)
  // パラメタを用いて proc の計算をする
  out procreply(ret, 返値)
end
```

クライアント側は次のようにします。

```
id = 自分の番号
out proc(id, パラメタ, ...)
in procreply(id, ?x) // x に値が返される
```

つまり、ret に各クライアントごとに違う番号を使うことで、他人と答えがごちゃまぜにならずに済むわけです。

3.2 Linda のおもちゃ

Linda は分散計算に関心のある人たちの興味を惹き、多くの実装が作られました。ここでも Java RMI を使って簡単な TS サーバを作ってみましょう (ただし 1 箇所のサーバで全部管理するので、スケーラビリティは全くありませんが)。まず、例によって RMI 用にサーバのインタフェースを作ります。

```
import java.rmi.*;

interface Sample45IF extends Remote {
  public void out(String key, String val) throws RemoteException;
  public String in(String key) throws RemoteException;
  public String read(String key) throws RemoteException;
}
```

簡単にしたので、タプルはキーと文字列の値 1 つだけを持ち、パターンマッチは実装しません。では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 Sample45Server extends UnicastRemoteObject implements Sample45IF {
  HashMap<String,String> tbl = new HashMap<String,String>();
  public Sample45Server() throws RemoteException { }
  public synchronized void out(String key, String val) {
    tbl.put(key, val); notifyAll();
  }
  public synchronized String in(String key) {
    while(true) {
      if(tbl.containsKey(key)) { break; }
```

```
      try { wait(); } catch(Exception ex) { }
    }
    String val = tbl.get(key); tbl.remove(key); return val;
  }
  public synchronized String read(String key) {
    while(true) {
      if(tbl.containsKey(key)) { break; }
      try { wait(); } catch(Exception ex) { }
    }
    String val = tbl.get(key); return val;
  }
  public synchronized void show() {
    for(String key: tbl.keySet()) {
      System.out.printf("%s: %s\n", key, tbl.get(key));
    }
  }
  public static void main(String[] args) throws Exception {
    Scanner sc = new Scanner(System.in);
    Sample45Server srv = new Sample45Server();
    Registry reg = LocateRegistry.createRegistry(4918);
    reg.bind("ts", srv);
    System.out.print("command> ");
    while(true) {
      String cmd = sc.next();
      if(cmd.equals("bye")) { break; }
      if(cmd.equals("show")) { srv.show(); }
      if(cmd.equals("out")) { srv.out(sc.next(), sc.next()); }
      sc.nextLine(); System.out.print("command> ");
    }
    System.exit(0);
  }
}
```

ここまできて今更ですが、RMIのサーバ内での呼び出しは各クライアント毎に別スレッドで実行されるので、synchronizedにより共有データに対する同期と保護を行います。そしてその中で、もし読みたいキーが表に入っていなければwait()により誰かが入れてくれるのを繰り返し待ちます。outを実行した側では、値を入れたのでとにかく待っている人を全員起こしてあげます。あと、サーバコマンドの側では現在のタプルを見たり、指定したタプルを追加する機能を入れてあります。

では次にクライアントとして、まずカウンタを作ってみます。

```
import java.rmi.*;

public class Sample45Countup {
  public static void main(String args[]) throws Exception {
    Sample45IF srv = (Sample45IF)Naming.lookup(args[0]);
    String x = srv.in("count");
    int v = Integer.parseInt(x) + 1;
    srv.out("count", "" + v);
    System.out.println(v);
```

```
    }
  }
```

このプログラムを 1 回実行するごとにカウンタが 1 増えます。ただし最初にサーバに count タプルを入れておく必要があることに注意。では次に、時刻サーバを作ってみます。

```
import java.rmi.*;

public class Sample45Clockserver {
  public static void main(String args[]) throws Exception {
    Sample45IF srv = (Sample45IF)Naming.lookup(args[0]);
    int time = 1;
    while(true) {
      srv.out("time", "" + time++);
      try { Thread.sleep(1000); } catch(Exception ex) { }
      srv.in("time");
    }
  }
}
```

最後に色々遊べるように、in/out/read の種別とタプルを自分で好きに打ち込めるようなクライアントを作ってみました。

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

public class Sample45Generic {
  public static void main(String args[]) throws Exception {
    Sample45IF srv = (Sample45IF)Naming.lookup(args[0]);
    Scanner sc = new Scanner(System.in);
    while(true) {
      System.out.print("command> ");
      String cmd = sc.next();
      if(cmd.equals("bye")) {
        System.exit(0);
      } else if(cmd.equals("out")) {
        srv.out(sc.next(), sc.next());
      } else if(cmd.equals("in")) {
        System.out.println(srv.in(sc.next()));
      } else if(cmd.equals("read")) {
        System.out.println(srv.read(sc.next()));
      }
      sc.nextLine();
    }
  }
}
```

これを使って時刻を見たりカウンタの現在値を読んだりできます。

演習 9 TS サーバを利用して、何か面白いことをやる例題を作ってみなさい。

3.3 Linda のその後

Linda については先に記したように多くの人が興味を持ち、実装も多く作られました。たとえば Sun は JavaSpaces というタプルスペース実装を作って公開していました。またその後、Sun は Jini というタプルスペースのモデルに基づいた通信サービスを策定し、ローカルな環境での情報共有 (たとえば家電の情報をホームネットワークで取って通信するなど) に使おうとしたりしていたのですが、実用に使おうと思うと TS をどのように安全にするか、ゴミ集めはどうするか、などの問題がいろいろあり、思ったように簡単にはできないことが分かってもたもたしているうちに下火になってしまいました。そしてそのうち Sun は Oracle に買収され…(以下略)。

まあそうなのですが、現在でもさまざまな人が作った TS の実装はありますし、今回やったように遊んでみるのなら面白いモデルだと思います。そしてそれなりに著名なので知っておいて損はないはずです。

4 最後に

本講義では、オブジェクト指向言語の基本的なメカニズムや機能からはじまって、その使い方、メタプログラミングやリフレクション、言語の記述とそのためのツール、分散のための機能やモデルなどを扱って来ました。お楽しみ頂けたでしょうか。

最後にレポート課題ですが、「これまでの各回に出ていた演習問題のどれでもいいので 1 つ選び、何かやってレポートにまとめ、提出する」ことをお願いしています。簡単な問題でも構いませんので。ただしレポートですので、次の内容は含まれること。

- 表紙 (「オブジェクト指向技術 レポート」、学籍番号、氏名、日付)
- 選んだ課題とその説明。
- 何をどのようにやってみたか。全体的な方針。
- コードを作ったり手直ししてみたはずなので、そのコードと直したりした内容の説明。
- 結果。
- 考察 (やってみて分かったこと)。
- 感想、その他

期限は「6 月一杯」としていますが、早めにやった方が忘れないでいいと思います。ではよろしく。