計算機プログラミング'99　＃8

久野　靖*

1999.10.20

0　はじめに

今回はまず「オブジェクトをそのままの形でファイルに書いたりネットワーク経由で転送する」直列化機能について学びます。次に、この機能を土台として実現されている「遠隔メソッド起動」と分散システムについて学びます。

1　直列化機能

「直列化」(serialization) とは、Java の言葉で「あるオブジェクトをバイトの列に変換 (してファイルやネットワークに送り込めるように) する」ことを言う。「直列化」というと別の意味にも取られ得るので困るのだけれど、もっと一般的な用語づかいでは「バイト列への符号化 (encoding)」などと呼ぶが長いし他にぴったりした用語がない。[1]

たとえば、あるクラスが「直列化可能」であるなら、そのクラスのオブジェクト (インスタンス) をバイト列に変換してファイルに書き込んだり、ネットワークストリーム経由でよそのマシンに転送したりできる (当然ながら、後でファイルを読み込んだり、転送されて受け取る側で、元と同じ値が復元できる)。

では、あるクラスを「直列化可能」にするにはどうしたらいいのだろう? クラスが直列化可能である条件は次の通り:

- そのクラスが implements java.io.Serializable であること。
- そのクラスのインスタンス変数の型が直列化可能であるか、または基本型であること。
- もし直列化可能でない型のインスタンス変数があるなら、その変数は transient(保存されない) と指定されていること。

つまり、まず「このクラスは直列化可能ですよ」という目印をつける (implements Serializable) ことが必要で、なおかつ「ファイルに書き出せない」要素を中に持っていないか、またはそのような要素は保存しないでよいと指定されていることが条件ということ。

なお、配列や Vector などの「いれもの」のクラスはその中に入れたものが直列化可能ならば、全体としても直列化可能になる。

では、どういうものを直列化可能にしたいだろう? たとえば、簡単なデータベースのようなプログラムを作るとして、そのデータをオブジェクトで表し、オブジェクトをそのままファイルに格納しておけると便利でしょう? 以下ではその例題用に次のクラスを用いる。

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

*筑波大学大学院経営システム科学専攻

[1]ちなみにデータベースや並列処理の世界では「複数の動作 A、B、C 等を同時に (並列に) 実行しても順番に (直列に) 実行した場合と同じ結果になる」ことを「直列化可能」という。全然違う意味なので混同しないように。

```
public class R8DataObj implements Serializable {
  String name, comm;
  int age;
  public R8DataObj(String n, int a, String c) { name = n; age = a; comm = c; }
  public String getName() { return name; }
  public String getComment() { return comm; }
  public int getAge() { return age; }
  public void setComment(String s) { comm = s; }
  public void setAge(int i) { age = i; }
}
```

では Serializable というのはどういうメソッドを定義しているのか、というと実はメソッドは「まったく定義していない」。つまり、Serializable というのは「このオブジェクトは直列化しますよ」ということを表すためだけに使うインタフェース。Java ではこのように、インタフェースを使って「このクラスはどういう性質を持つ」ということをシステムに教えるようになっている。

直列化可能なオブジェクトをファイルなどに読み書きするには、ObjectInputStream や ObjectOutputStream を作り、そのメソッド readObject()、writeObject() を使えばよい。では、上のクラスを Vector に入れた状態でそっくり読み書きすることで、簡単なデータベースのようなものを作る例題を見てみよう。

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

public class R8Sample1 {
  public static void main(String[] args) {
    Vector db = new Vector();
    try {
      ObjectInputStream oi = new ObjectInputStream(
                              new FileInputStream("obj.data"));
      db = (Vector)oi.readObject(); oi.close();
    } catch(Exception e) {
      System.out.println("Cannot read obj.data; create new database");
    }
    try {
      BufferedReader in=new BufferedReader(new InputStreamReader(System.in));
      while(true) {
        System.out.print("command> "); System.out.flush();
        String line = in.readLine();
        if(line.equals("quit")) {
          break;
        } else if(line.equals("add")) {
          System.out.print("name: "); System.out.flush();
          String name = in.readLine();
          System.out.print("age: "); System.out.flush();
          int age = (new Integer(in.readLine())).intValue();
          System.out.print("comment: "); System.out.flush();
          String comm = in.readLine();
          db.add(new R8DataObj(name, age, comm));
        } else if(line.equals("show")) {
          System.out.print("name: "); System.out.flush();
```

```
            String name = in.readLine();
            for(Enumeration e = db.elements(); e.hasMoreElements(); ) {
              R8DataObj obj = (R8DataObj)e.nextElement();
              if(obj.getName().equals(name)) {
                System.out.println("name: " + obj.getName());
                System.out.println("age: " + obj.getAge());
                System.out.println("comment: " + obj.getComment());
              }
            }
          } else { System.out.println("?" + line); }
        }
      } catch(Exception e) {
        System.out.println("error: "+e); System.out.println("saving...");
      }
      try {
        ObjectOutputStream oo = new ObjectOutputStream(
                                  new FileOutputStream("obj.data"));
        oo.writeObject(db); oo.close();
      } catch(Exception e) {
        System.out.println("error: "+e);
      }
    }
  }
```

「バイト単位でのファイルの読み書き」は前にやった FileInputStream、FileOutputStream でできるので、これをもとに ObjectInputStream、ObjectOutputStream を生成すればファイルに対してオブジェクトが読み書きできる。ループの内側でのコマンド処理は別に目新しいことはない (が、内部でエラーが起きた時は try-catch で受け止めてデータを保存して終るようになっていることに注意。

演習 1 この例題をコピーしてきてそのまま動かし、適当なデータベースを生成せよ。

演習 2 この例題で作ったデータベースに対して次の操作を行うようなプログラムを作ってみよ。

a. データベースの内容全部をそのまま見える形で表示する。
b. データベース中の全データの「年齢」を 1 ふやす。
c. データベースから指定した名前のデータを削除する。名前はコマンド引数で「java R8Ex8 kuno」のようにして指定する簡単でよい (プログラムの中からは `args[0]` で参照できる)。
d. データベース中のすべての人に対して、その人の「クローン」のデータを追加する。ただしクローンの名前は元の名前の後に「1」を追加したもの (たとえば kuno1 のようにする)、年齢も元の人の名前に「1」を足したもの。

2 データベースクラスの分離

上のコードでは、R8DataObj を保存したり管理する部分と、データを検索したり追加する部分が「ごちゃまぜ」で美しくない。データを管理する部分を R8DBService という名前で分離してみた。なお、このクラスでは getObj() で取り出したオブジェクトはデータベースから削除されるので、更新したい場合は変更して再度 putObj() する必要がある。

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

```
import java.util.*;

public class R8DBService {
  Vector db = new Vector();
  public void load(String s) {
    try {
      ObjectInputStream oi = new ObjectInputStream(new FileInputStream(s));
      db = (Vector)oi.readObject(); oi.close();
    } catch(Exception e) { System.err.println("Cannot load DB from " + s); }
  }
  public R8DataObj getObj(String name) {
    for(Enumeration e = db.elements(); e.hasMoreElements(); ) {
      R8DataObj o = (R8DataObj)e.nextElement();
      if(o.getName().equals(name)) { db.removeElement(o); return o; }
    }
    return null;
  }
  public void putObj(R8DataObj obj) {
    db.addElement(obj);
  }
  public String[] getNames() {
    String[] a = new String[db.size()];
    for(int i = 0; i < db.size(); ++i) {
      a[i] = ((R8DataObj)db.elementAt(i)).getName();
    }
    return a;
  }
  public void save(String s) {
    try {
      ObjectOutputStream oo = new ObjectOutputStream(new FileOutputStream(s));
      oo.writeObject(db); oo.close();
    } catch(Exception e) { System.err.println("cannot save DB to " + s); }
  }
}
```

演習 3 先の例題や、演習 2 で作ったプログラムを、このクラスを利用する形に書き直してみよ。

3 アプレットでない画面プログラム

さて次に、これまでグラフィクスや GUI を使うプログラムはすべてアプレットだったが、これをアプレットでないプログラム (スタンドアロンアプリケーション) でやるにはどうするか、という質問を頂いているのでその例題をやっておこう。まず、アプレットではウィンドウはブラウザが用意してくれていたが、スタンドアロンアプリケーションでは自分でウィンドウを作り出さなければならない。ウィンドウを表すクラスは Frame なので、基本的には次のようにすればよい。

```
Frame fr = new Frame();  // ウィンドウを用意
fr.setSize(400, 300);    // 大きさ (幅と高さ) を指定
fr.setVisible(true);     // 画面に表示
```

しかし、これだけでは「中身のない」ウィンドウができてそれっきりである。ではなくて、これまでアプレットでやったようにしたければどうしたらいいだろう? それには、Frame のサブクラスを作って、その中はアプレットと同様に作成し、上記の Frame の代りにこのクラスを使うようにすればよい。もちろん、部品の割り付けなどは init() に入れるだろうから、上記の 3 行の後で init() を呼んでやる必要がある。実際、そのようなクラスを作ってみよう。まず、main() の側は上で述べたようなこと (とデータベースを生成して渡すこと) しかしない。

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;

public class R8Sample2 {
  public static void main(String[] args) {
    final R8DBService db = new R8DBService(); db.load("obj.data");
    R8Frame fr = new R8Frame(db);
    fr.setSize(400, 300); fr.setVisible(true); fr.init();
    fr.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent evt) {
        db.save("obj.data"); System.exit(0);
      }
    });
  }
}
```

なお、このプログラムでは作成したウィンドウを閉じる (Windows98/NT では「×」ボタンを押す、X11 ではウィンドウマネージャの Delete 機能でその窓を消す) とデータベースを保存して終了するようになっているので、そのためのイベントアダプタを窓にたいして付加している。

では、窓のクラスを見てみよう。ほとんどこれまでやってきたアプレットと変わらないことがわかる。

```
class R8Frame extends Frame {
  R8DBService db;
  TextField t0 = new TextField();
  TextField t1 = new TextField();
  TextArea a0  = new TextArea();
  Button b0 = new Button("Retrieve");
  Button b1 = new Button("Add");
  Button b2 = new Button("List");
  List l1 = new List();
  TextField l0 = new TextField();
  public R8Frame(R8DBService d) { db = d; }
  public void init() {
    setLayout(null);
    add(t0); t0.setBounds(10, 40, 100, 30);
    add(t1); t1.setBounds(110, 40, 60, 30);
    add(a0); a0.setBounds(10, 80, 240, 120);
    add(l1); l1.setBounds(260, 40, 130, 200);
    add(b0); b0.setBounds(10, 200, 60, 40);
    add(b1); b1.setBounds(80, 200, 60, 40);
    add(b2); b2.setBounds(150, 200, 60, 40);
```

```
    add(l0); l0.setBounds(10, 240, 380, 30);
    b0.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          R8DataObj o = db.getObj(t0.getText());
          if(o == null) { l0.setText("Not Found"); return; }
          t1.setText((new Integer(o.getAge())).toString());
          a0.setText(o.getComment());
        } catch(Exception e) { l0.setText(e.toString()); }
      }
    });
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          db.putObj(new R8DataObj(t0.getText(),
                    (new Integer(t1.getText())).intValue(), a0.getText()));
          t0.setText(""); t1.setText(""); a0.setText("");
        } catch(Exception e) { l0.setText(e.toString()); }
      }
    });
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          String[] a = db.getNames();
          l1.removeAll();
          for(int i = 0; i < a.length; ++i) l1.add(a[i]);
        } catch(Exception e) { l0.setText(e.toString()); }
      }
    });
  }
}
```

演習 4 このプログラムをコピーしてきてそのまま動かせ。

演習 5 ブラウザにたよらず自分で窓を作るプログラムを、これまでに作ったアプレットを改造したり、新しく設計して作成せよ。

4 分散システムと RMI

ここまででは、あくまでも 1 つのプログラムがファイルから読み、画面の窓でそれを操作し、終ったらファイルに書き出していた。しかし実際にはこのような作業は、「中央にデータベースがあり」「各地からネットワーク経由でこれを操作する」という形のアプリケーションになるのが普通である。それにはどうしたらいいだろう? 昔であれば「DBMS を使う」というのがまず考え付いたのだが、現在では「クライアントサーバ (C/S) システムにする」というのがもう 1 つの (しかも作りやすい) 解になっている。というのは、中央のサーバプログラムはデータを一括して管理することができるからである。

しかし、従来の C/S システムはサーバとクライアントがネットワーク経由の通信路を張って、その後は InputStream/OutputStream のようなもので相互にやりとりしながら協調作業を進めていくので、開発は結構面倒だった。

これに対し、最近は「分散オブジェクト技術」を用いることによって、ストリームのようなものを介在せずに遠隔地のオブジェクトを「直接アクセスする」(かのように見せかける) 形でシステムが開発できるようになってきた。ここで、遠隔地にあるオブジェクトとの通信を橋渡ししてくれる機能を ORB(Object Request Broker) という。Java で使われる ORB としては

- CORBA — 言語独立なので、Java 以外の言語、たとえば C++や Smalltalk-80 などと組み合わせても使える。
- HORB — 電総研の平野氏が開発した、国産の Java 専用 ORB。Java RMI より先行したので結構普及した。
- Java RMI — Sun が開発した Java 標準 API に含まれる ORB。

がある。ここでは Java RMI を使ってみることにする。なお、RMI とは Remote Method Invocation、つまり遠隔地にあるオブジェクトのメソッド (method) をネットワーク経由で (remote から) 起動する (invocation)、という意味。

ところで、アプレットではご存じの通り (だったっけ?)、ファイルの読み書きやネットワーク接続が禁止されているが、ただしアプレットを取り寄せて来た元のサーバ (我々のシステムでは smm) には接続することができるので、C/S システムのサーバも smm で動かせばアプレットから使うことができる。こうすれば、クライアントはアプレットだからどこででも簡単に立ち上げられ、データの管理は手元のサーバで行えるという分散システムが簡単に作れる。

ではごたくはそれくらいにして、RMI を使うための説明をはじめる。まず、遠隔地から使うオブジェクトは必ず Remote インタフェース (Remote のサブインタフェース) の変数に格納して利用することになっている。このためのインタフェース定義を示す。

```
import java.rmi.*;

public interface R8RemoteDB extends Remote {
  public R8DataObj getObj(String name) throws RemoteException;
  public void putObj(R8DataObj obj) throws RemoteException;
  public String[] getNames() throws RemoteException;
}
```

これはもちろん、さっきの R8DBService を遠隔地から使うためのインタフェースだが、遠隔地からではデータベースのロード/セーブはさせないので、そのようなメソッドは除いてある。また、各メソッドは実際に呼ぼうとしたときネットワークの不調などがあると RemoteException を投げるので、それを宣言しておく必要がある。

次に、このインタフェースを実装するクラスを示そう。

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

public class R8DBServiceRemote
    extends UnicastRemoteObject implements R8RemoteDB {
  R8DBService db;
  public R8DBServiceRemote() throws RemoteException {
    db = new R8DBService();
  }
  public void load(String s) { db.load(s); }
  public R8DataObj getObj(String name) { return db.getObj(name); }
  public void putObj(R8DataObj obj) { db.putObj(obj); }
  public String[] getNames() { return db.getNames(); }
```

```
  public void save(String s) { db.save(s); }
}
```

このオブジェクトは遠隔地からメソッドを呼び出されるようにするので、そのための機能を UnicastRemoteObject クラスから継承する。また、コンストラクタではオブジェクト生成時にネットワーク接続がうまく行かない場合があるため、RemoteException を投げると宣言する必要がある。

残りの部分については、既に作った R8DBService クラスの機能をそのまま呼んでいるだけである (ロード/セーブもある)。本当は継承で済ませたいのだけど、既に UnicastRemoteObject から継承しているためそれはできない。

さて、これらをコンパイルするのだが、ちょっと追加の「おまじない」が必要になる。

```
% javac R8RemoteDB.java          ←普通
% javac R8DBServiceRemote.java  ←普通
% export CLASSPATH=.            ← rmic コマンドのためになぜか必要
% rmic  R8DBServiceRemote       ←スタブ/スケルトンを生成
```

最後の「rmic」コマンドはスタブとスケルトンという小さなクラスを生成する。スタブとは、遠隔側で「本物」の代りに呼び出されて、呼び出しをネットワーク通信に変換するための橋渡しクラス。スケルトンとは、サーバ側でスタブと通信してそれに従って「本物」を呼び出すための橋渡しクラス。これらはソースコードは不要で、rmic によっていきなり.class ファイルが生成される。

さて、実際にこれを遠隔地から呼ぶためには、サーバ側で動作するメインプログラムが必要である。

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

public class R8Sample3Server {
  public static void main(String[] args) {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    try {
      R8DBServiceRemote db = new R8DBServiceRemote();
      db.load("obj.data");
      Registry reg = LocateRegistry.createRegistry(2020);
      reg.bind("DB", db);
      System.out.println("Bind Complete");
      while(!in.readLine().equals("quit")) { }
      System.out.println("Saving Data...");
      db.save("obj.data");
      System.exit(0);
    } catch(Exception e) { e.printStackTrace(); }
  }
}
```

このメインプログラムは、まずデータベースを用意して、次にレジストリ (というのは遠隔側からオブジェクトを探索に来たときに応答するための「オブジェクトのデータベース」) を生成する。通信は TCP/IP によるので、ポート番号を指定する必要がある。本日は他人と衝突しないように「2000+自分のユーザ ID」を使うことにしよう。そして、レジストリに対して遠隔参照されるオブジェクトを登録すれば、あとは遠隔側からオブジェクトを参照してメソッドを呼び出せるようになる。

なお、最後にサーバを終らせてデータベースを保存する必要があるので、その後繰り返しキーボードから 1 行読み込み、「quit」と入力されたらデータを保存してプログラムを終らせるようになっている。

では最後に、アプレットクラスを示す (画面の大きさを 400 × 300 で設計したので注意)。

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.rmi.*;

public class R8Sample3 extends Applet {
  R8RemoteDB db;
  TextField t0 = new TextField();
  TextField t1 = new TextField();
  TextArea a0  = new TextArea();
  Button b0 = new Button("Retrieve");
  Button b1 = new Button("Add");
  Button b2 = new Button("List");
  List l1 = new List();
  TextField l0 = new TextField();
  public void init() {
    setLayout(null);
    add(t0); t0.setBounds(10, 40, 100, 30);
    add(t1); t1.setBounds(110, 40, 60, 30);
    add(a0); a0.setBounds(10, 80, 240, 120);
    add(l1); l1.setBounds(260, 40, 130, 200);
    add(b0); b0.setBounds(10, 200, 60, 40);
    add(b1); b1.setBounds(80, 200, 60, 40);
    add(b2); b2.setBounds(150, 200, 60, 40);
    add(l0); l0.setBounds(10, 240, 380, 30);
    b0.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          R8DataObj o = db.getObj(t0.getText());
          if(o == null) { l0.setText("Not Found"); return; }
          t1.setText((new Integer(o.getAge())).toString());
          a0.setText(o.getComment());
        } catch(Exception e) { l0.setText(e.toString()); }
      }
    });
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          db.putObj(new R8DataObj(t0.getText(),
                    (new Integer(t1.getText())).intValue(), a0.getText()));
          t0.setText(""); t1.setText(""); a0.setText("");
        } catch(Exception e) { l0.setText(e.toString()); }
      }
    });
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
```

```
        try {
          String[] a = db.getNames();
          l1.removeAll();
          for(int i = 0; i < a.length; ++i) l1.add(a[i]);
        } catch(Exception e) { l0.setText(e.toString()); }
      }
    });
    try {
      db = (R8RemoteDB)Naming.lookup("rmi://smm:2020/DB");
    } catch(Exception e) { l0.setText("?" + e.toString()); }
  }
}
```

アプレットはさっきの R8Frame と「ほとんど」同じであるが、ただしコンストラクタでデータベースを受け取る代りに init() の最後でレジストリに問い合わせてデータベースを受け取るところが違う。ここでポート番号やデータベース名称はさっきサーバで指定したものと同じでなければならない。さらに、ホスト名は smm でなければならない (アプレットの制約のため)。

これで、さっきと同じコードがアプレット内で動くのだけど、実は今度はデータベースオブジェクトは「どっかネットワークの向こう」にあって、そこのメソッドを呼び出しているということが本質的に違っている。たとえば複数のユーザが同一サーバを利用した場合、当然その変更内容は全員に共有される。

さて、今回最初に直列化をやった訳がわかりますか? 実は、こうして RMI 経由でメソッドを呼び出す時は、引数や返値は直列化されてネットワーク経由で転送されている (考えてみれば当り前)。ということは、そのつどオブジェクトのコピーができてしまうので、副作用や共有に関してこれまでのローカルな動作と違う点もあることは注意する必要がある。

演習 6 このクラス群をコピーしてきてポート番号は自分で書き換え、そのまま動かしてみよ。動いたら、他人が動かしているサーバのポート番号に取り換えてみて、データが共有されていることを確かめよ。

演習 7 これを参考に、自分なりに RMI を活用したプログラムを作ってみよ。

A 期末課題

既に先週から「次週までの課題」を出していませんが (はっきり言うのを忘れてすいません!)、今後とも同様です。その代り、そろそろ最終課題の準備に入ってください。

> **課題:** 自分の腕前にあった、興味深い Java プログラム (アプレットでもアプリケーションでもよい) を作成し、報告せよ。

最終レポートはいつものような感想だけでなく、次の構成できちんと作成してください

1. 表紙。「計算機プログラミング 1999 年度 期末レポート」とタイトルをつけ、自分の学籍番号、氏名、提出日時を明記する。
2. テーマ。どのようなプログラムを作成したか、「仕様」を書く。
2. 方針。どのような方針で上記のテーマを決めたかを書く。
3. 設計。プログラムの設計を説明する。
4. プログラムのリスティングとその説明。
5. 実行例。対話例や画面ダンプなど。
6. 考察。この課題でどんなことが分かったか、また残っている疑問点などを整理し考察する。考察は感想ではない。
7. 最後に、感想や講義全体に対するご意見などを自由にお願いします。